Ph.D. Dissertation

Architectural Techniques for Memory Systems based on Emerging Memory Technologies

새로운 메모리 기술을 기반으로 한 메모리 시스템 설계 기술

February 2017

Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

Junwhan Ahn

Architectural Techniques for Memory Systems based on Emerging Memory Technologies

새로운 메모리 기술을 기반으로 한 메모리 시스템 설계 기술

지도교수 최기영

이 논문을 공학박사 학위논문으로 제출함

2016년 11월

서울대학교 대학원

전기·컴퓨터공학부

안준환

안준환의 공학박사 학위논문을 인준함

2016년 12월

위 원 장	이혁재	(인)
부위원장	최기영	(인)
위 원	유승주	(인)
위 원	이재욱	(인)
위 원	김장우	(인)

Abstract

Architectural Techniques for Memory Systems based on Emerging Memory Technologies

Junwhan Ahn
Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

Performance and energy efficiency of modern computer systems are largely dominated by the memory system. This memory bottleneck has been exacerbated in the past few years with (1) architectural innovations for improving the efficiency of computation units (e.g., chip multiprocessors), which shift the major cause of inefficiency from processors to memory, and (2) the emergence of data-intensive applications, which demands a large capacity of main memory and an excessive amount of memory bandwidth to efficiently handle such workloads. In order to address this *memory wall* challenge, this dissertation aims at exploring the potential of emerging memory technologies and designing a high-performance, energy-efficient memory hierarchy that is aware of and leverages the characteristics of such new memory technologies.

The first part of this dissertation focuses on energy-efficient on-chip cache design based on a new non-volatile memory technology called *Spin-Transfer Torque RAM (STT-RAM)*. When STT-RAM is used to build on-chip caches, it provides several advantages over conventional charge-based memory (e.g., SRAM or eDRAM), such as non-volatility, lower static power, and higher density. However, simply replacing SRAM caches with STT-RAM rather increases the energy consumption because write operations of STT-RAM are slower and more energy-consuming than those of SRAM.

To address this challenge, we propose four novel architectural techniques that can alleviate the impact of inefficient STT-RAM write operations on system performance and energy consumption. First, we apply STT-RAM to instruction caches (where write operations are relatively infrequent) and devise a power-gating mechanism called *LASIC*, which leverages the non-volatility of STT-RAM to turn off STT-RAM instruction caches inside small loops. Second, we propose *lower-bits cache*, which exploits the narrow

bit-width characteristics of application data by caching frequent bit-flips at lower bits in a small SRAM cache. Third, we present *prediction hybrid cache*, an SRAM/STT-RAM hybrid cache whose block placement between SRAM and STT-RAM is determined by predicting the write intensity of each cache block with a new hardware structure called write intensity predictor. Fourth, we propose *DASCA*, which predicts write operations that can bypass the cache without incurring extra cache misses (called dead writes) and lets the last-level cache bypass such dead writes to reduce write energy consumption.

The second part of this dissertation architects intelligent main memory and its host architecture support based on *logic-enabled DRAM*. Traditionally, main memory has served the sole purpose of storing data because the extra manufacturing cost of implementing rich functionality (e.g., computation) on a DRAM die was unacceptably high. However, the advent of 3D die stacking now provides a practical, cost-effective way to integrate complex logic circuits into main memory, thereby opening up the possibilities for *intelligent* main memory. For example, it can be utilized to implement advanced memory management features (e.g., scheduling, power management, etc.) inside memory; it can be also used to offload computation to main memory, which allows us to overcome the memory bandwidth bottleneck caused by narrow off-chip channels (commonly known as *processing-in-memory* or PIM). The remaining questions are what to implement inside main memory and how to integrate and expose such new features to existing systems.

In order to answer these questions, we propose four system designs that utilize logic-enabled DRAM to improve system performance and energy efficiency. First, we utilize the existing logic layer of a Hybrid Memory Cube (a commercial logic-enabled DRAM product) to (1) dynamically turn off some of its off-chip links by monitoring the actual bandwidth demand and (2) integrate prefetch buffer into main memory to perform aggressive prefetching without consuming off-chip link bandwidth. Second, we propose a scalable accelerator for large-scale graph processing called *Tesseract*, in which graph processing computation is offloaded to specialized processors inside main memory in order to achieve memory-capacity-proportional performance. Third, we design a low-overhead PIM architecture for near-term adoption called *PIM-enabled instructions*, where PIM operations are interfaced as cache-coherent, virtually-addressed host processor instructions that can be executed either by the host processor or in main memory depending on the data locality. Fourth, we propose an energy-efficient PIM

system called *aggregation-in-memory*, which can adaptively execute PIM operations at any level of the memory hierarchy and provides a fully automated compiler toolchain that transforms existing applications to use PIM operations without programmer intervention.

Keywords: Computer Architecture, Memory Hierarchy, Non-Volatile Memory, STT-RAM, Logic-Enabled DRAM, Processing-in-Memory
Student Number: 2011-20873

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of the *big-data* era, which consists of increasingly data-intensive workloads and continuous supply and demand for more data and their analyses, the design of computer systems for efficiently processing large amounts of data has drawn great attention. From the data storage perspective, the current realization of big-data processing is based mostly on secondary storage such as hard disk drives and solid-state drives. However, the continuous effort on improving cost and density of DRAM opens up the possibility of *in-memory* big-data processing. Storing data in main memory achieves orders of magnitude speedup in accessing data compared to conventional disk-based systems, while providing up to terabytes of memory capacity per server. The potential of such an approach in data analytics has been confirmed by both academic and industrial projects, including RAMCloud [1], Pregel [2], GraphLab [3], Oracle TimesTen [4], and SAP HANA [5].

While the software stack for in-memory big-data processing has evolved, developing a hardware system that efficiently handles a large amount of data in main memory still remains as an open question. In particular, the efficiency of a computer system that processes a large amount of data is determined primarily by *how fast and efficient the massive amount of data can be delivered from memory to computation units*. Unfortunately, this memory bottleneck has been aggravated since computational power has been continuously increasing through architectural innovations (e.g., chip multiprocessors, specialized accelerators, etc.), whereas the memory performance and energy efficiency

cannot be easily improved due to the inherent limitations in its material characteristics, narrow channels between processing units and memory, and so on.

1.1 Inefficiencies in the Current Memory Systems

1.1.1 On-Chip Caches

Many traditional workloads exhibit a good amount of temporal and spatial data locality. For such workloads, last-level caches play an important role in bridging the gap between processor and memory speeds. As more cores are expected to be integrated on a chip, it is inevitable to use larger last-level caches to meet the ever-increasing bandwidth demand from multiple cores. For example, an Intel Ivytown Xeon processor [6] equips a 37.5 MB L3 cache, which takes a substantial portion of the die area.

However, enlarging the last-level caches is hitting its limit mainly due to the power-hungry conventional memory technologies such as SRAM or embedded DRAM (eDRAM). Due to their charge-based nature, they consume a significant amount of background power (e.g., leakage power of SRAM and refresh power of eDRAM), which increases as the cache capacity becomes larger and/or the process technology scales down. This implies that the energy cost of just keeping on-chip cache data alive is increasing, which greatly degrades the energy efficiency of large caches.

On top of that, limited scalability of traditional memory technologies makes building large on-chip caches even more challenging in deep submicron technologies. Traditional charge-based memory technologies rely on the difference in the number of electrons in a cell to distinguish logical '0' and '1' states. However, as the number of electrons that can be held in a single memory cell decreases with the technology shrinking, it has become more and more challenging to reduce the size of a memory cell without affecting the reliable distinction between the two logical states.

1.1.2 Main Memory

For workloads whose working set size exceeds the last-level cache capacity, the performance and the energy efficiency of such workloads are mainly determined by main memory. This situation happens in many in-memory data analytics workloads, which process a large amount of data in main memory. For example, large-scale graph processing (widely used in social network analysis, web search engines, machine learning, etc.) is known to be memory-bandwidth-bound due to its large memory footprint and low data locality in memory accesses [7].

However, providing high memory bandwidth in a practical, cost-effective manner is challenging under the conventional memory hierarchy design. In today's memory hierarchy, main memory bandwidth is limited by narrow off-chip channels between the CPU and the main memory and this has widened the discrepancy between the computation speed and the data transfer speed (commonly known as the *memory wall* [8]). For example, Intel Xeon E5-2600 v3 [9] equips four channels of DDR4-2133, which provides up to 68 GB/s in total. This is more than an order of magnitude lower than its last-level cache bandwidth, which is 2.1 TB/s at the maximum frequency. Thus, applications with large working sets are much easier to be bounded by memory bandwidth, thereby implying the importance of main memory systems that can provide high bandwidth to computation units.

What is worse, main memory bandwidth does not scale well under the current memory hierarchy organization, thereby creating the *bandwidth wall* [10]. This is because the CPU and the main memory communicate with each other via off-chip channels, and thus, increasing the main memory bandwidth requires more I/O pins per CPU socket under the same I/O speed per channel. Since high-end CPU sockets already have numerous pins (e.g., 2011 pins in Intel Xeon E5-2600 v3 [9]), adding more memory channels to each CPU socket (which requires more than 100 pins per DDR3/4 channel) is considered as impractical from the cost-effectiveness perspective. In fact, Intel maintained the same pin count for the Xeon EP family across four generations from Sandy Bridge to Broadwell. This clearly shows the limited scalability of the number of main memory channels in the current memory hierarchy design.

1.2 New Memory Technologies: Opportunities and Challenges

1.2.1 Energy-Efficient On-Chip Caches based on STT-RAM

As explained previously, energy efficiency and scalability limitations of conventional on-chip caches come from the charge-based nature of underlying memory technologies (e.g., SRAM and eDRAM). This implies that such drawbacks could potentially be overcome by building on-chip caches with *non-charge-based* memory technologies that are fast enough to be used as on-chip caches.

Recently, non-volatile memory technologies have received lots of attention mainly due to its improved energy efficiency over conventional charge-based memory technologies. Among them, magnetoresistive RAM (MRAM) has been extensively studied for its possibility to be used primarily as working memory (i.e., on-chip caches and main memory) because of its attractive characteristics, including high performance, low

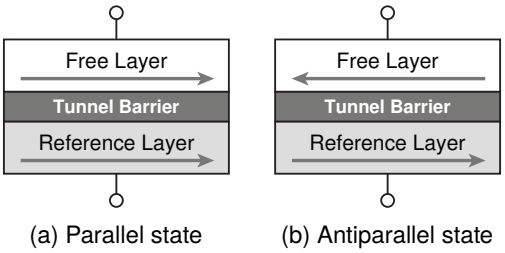


Figure 1.1: Magnetic tunnel junction of STT-RAM.

power consumption, high density, and nearly unlimited endurance. MRAM also provides good compatibility with standard CMOS processes, which contributes to lowering the manufacturing cost [11]. Due to these advantages, MRAM has been actively researched as a promising future memory technology by both academia and industry (e.g., Samsung Electronics, Hynix, Toshiba, Everspin Technologies, and Crocus Technology).

The key difference of MRAM against other memory technologies is that it stores the information as a form of magnetization direction. For this purpose, MRAM uses a new information storage called *magnetic tunnel junction (MTJ)*. An MTJ is composed of two ferromagnetic layers (free and reference layers) and one tunnel barrier between them. The magnetization direction of the free layer can be freely changed, while that of the reference layer is fixed at fabrication. The data bit stored in an MTJ is represented by the difference in magnetization direction between two ferromagnetic layers (on which the resistance of the MTJ depends), thereby creating two possible states: parallel and antiparallel states (shown in Figure 1.1). An MTJ in the parallel state shows lower resistance compared to the one in the antiparallel state. Since these states can be maintained without any power consumption, MRAM shows low leakage power consumption and non-volatility compared to conventional charge-based memory technologies.

In order to read the information from an MTJ, a read operation is performed by applying a small voltage difference between two electrodes of the MTJ and sensing the current flow. Since the parallel state and the antiparallel state have resistance difference, the sensing circuit can access the information stored in the MTJ by comparing the current flowing through the target MTJ and the reference MTJ.

There are two major mechanisms to write data to an MTJ. The first generation of the MRAM technology switches the MTJ state by using an external magnetic field generated by large current. However, this mechanism limits the scalability of write energy and

latency since larger write current is needed as the size of an MTJ becomes smaller. This limited scalability leads to the development of a new MRAM technology, called *Spin-Transfer Torque RAM (STT-RAM)* [12]. STT-RAM performs a write operation by directly applying a large voltage difference between two electrodes for a given duration, called write pulse width. Then, spin-polarized current flowing through the MTJ changes the magnetization direction of the free layer (based on a physical phenomenon called *spin-transfer torque*). For example, if the current flows from the reference layer to the free layer (e.g., upward direction in Figure 1.1), the reference layer acts as a spin filter, which filters out electrons whose spin direction is different from that of the reference layer (i.e., spin-polarizing the outgoing current). Then, the spin-polarized current magnetizes the free layer towards the magnetization direction of the reference layer. This new write mechanism scales well because the amount of required write current *decreases* as the size of an MTJ becomes smaller.

When an on-chip cache is built with STT-RAM, it provides several advantages over conventional SRAM caches. As a concrete example, Table 1.1 compares the characteristics of a 1 MB STT-RAM cache in 45 nm technology with its SRAM counterpart (modeled by CACTI 6.5 [13] and NVSim [14], refer to Section 6.4.1 for the detailed methodology). Based on the modeling result, the key advantages of STT-RAM over SRAM can be summarized as follows:

- **Non-volatility**: STT-RAM persists the stored data even without power supply.
- **Low static power**: Unlike SRAM, STT-RAM consumes very low static power due to its non-volatility. In a typical 1T-1MTJ STT-RAM cell (which is composed of one MTJ and one access transistor) [12], the access transistor is turned on only during the cell is accessed (similar to DRAM). This effectively makes static power consumption of STT-RAM cells close to zero, thereby leaving its peripheral circuits as the only source of static power consumption.
- **Higher density**: Due to its simpler cell structure (one magnetic tunnel junction and one access transistor), the density of STT-RAM is $\sim 4\times$ higher than SRAM.
- **High performance**: Unlike other resistive memory, read performance of STT-RAM is comparable to that of SRAM. For high-capacity caches, STT-RAM even shows slightly lower read latency than SRAM as shown in the table because its higher density contributes to reducing the global wire length.
- **Better scalability**: As mentioned previously, STT-RAM cells are inherently scalable because the read/write current decreases as the cell size becomes smaller.

Table 1.1: Characteristics of 1 MB SRAM and STT-RAM Caches

	SRAM	STT-RAM	Difference
Area (mm^2)	3.77	0.95	-74%
Read Latency (ns)	3.18	2.83	-11%
Write Latency (ns)	3.18	12.01	+278%
Read Energy (nJ)	0.08	0.07	-16%
Write Energy (nJ)	0.08	0.64	+670%
Static Power (mW)	14.63	2.32	-84%

- **CMOS compatibility**: STT-RAM can be embedded into standard CMOS manufacturing processes with just two extra masks [12].

While all of the aforementioned benefits are extremely favorable in constructing energy-efficient on-chip caches, STT-RAM also has a drawback in that its write latency and energy is several times higher than SRAM (e.g., 3x longer write latency and 7x higher write energy in Table 1.1). This inefficiency in write operations comes from the write mechanism of STT-RAM, which requires the voltage difference applied to the MTJ to be large enough to flip the magnetization direction of its free layer. What is worse, the asymmetry between read and write energy is expected to remain even in future STT-RAM technologies because the write current cannot be arbitrarily reduced since there is a minimum ratio of write to read current due to read disturbance [12] while there is a minimum level of read current to avoid read sensing errors [15]. As we will quantitatively show later in this dissertation, minimizing the impact of such inefficient write operations is the key to making STT-RAM caches attractive compared to conventional SRAM caches.

1.2.2 Intelligent Main Memory based on Logic-Enabled DRAM

Since both the memory wall and the bandwidth wall problems are because of narrow off-chip channels between the CPU and main memory, *placing the computation close to main memory* can help mitigate those problems. A representative effort towards this direction is called *processing-in-memory (PIM)*, where computation units are placed inside main memory to enable low-latency, high-bandwidth access to memory-resident data without being limited by narrow off-chip channels. This PIM concept was extensively studied by many researchers in 1990s, mostly in the form of systems that implement custom processors or reconfigurable logic on DRAM dies [16–22].

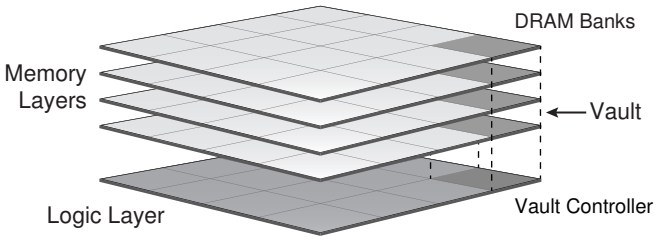


Figure 1.2: Structure of a hybrid memory cube.

However, despite the advantages of the PIM concept and a decade of research, PIM was not commercialized in the end. This is mainly due to the high manufacturing cost of integrating complex logic circuits and high-density DRAM. Nowadays, state-of-the-art DRAM manufacturing processes are significantly different from high-performance logic manufacturing processes in that the former are tailored to maximize the density of DRAM dies. Thus, complex logic circuits (such as general-purpose processors) implemented with DRAM processes not only operate at a lower speed than its equivalent logic process implementation but also unacceptably degrade the density of DRAM. The latter was particularly critical because the DRAM market has been generally driven by cost per GB, and thus, modifications that can significantly increase the manufacturing cost have been considered to be unfavorable by industry [23].

More than a decade after these initial efforts, PIM is recently regaining attention. This is because PIM has become a much more realizable, practical, and demanding technology than before due to the following two reasons. First, the emergence of 3D die stacking technologies provides cost-effective integration of complex logic and DRAM (e.g., Hybrid Memory Cube [24], JEDEC High-Bandwidth Memory [25], etc.). Second, the increasing demand for data-intensive computing (e.g., big-data workloads) motivates the need for a different computing paradigm that can provide scalable memory bandwidth to computation units.

For example, the Hybrid Memory Cube (HMC) [24] is a commercial 3D-stacked DRAM product from industry, where each cube is composed of one logic layer and multiple DRAM layers (see Figure 1.2). The DRAM layers of an HMC are connected to the logic layer with thousands of through-silicon vias (TSVs), thereby providing a few Tb/s of internal bandwidth, while the logic layer facilitates high-speed serial links between the CPU and HMCs achieving up to 320 GB/s of external bandwidth. Moreover, DRAM layers are designed to provide abundant parallelism inside an HMC through hundreds of banks per device [26].

In the HMC design, tight integration of a logic die and multiple DRAM dies brings the following key advantages and opportunities to memory system design:

- **Fast and efficient internal data transfer**: HMCs can be designed to provide much higher memory bandwidth inside a DRAM chip (between the logic die and the DRAM dies) than between the CPU and HMCs because TSVs are much more cost-effective than off-chip wires and package pins. Moreover, vertical data transfer inside an HMC between logic and memory is more energy-efficient than off-chip transfer due to the much shorter wire length.

- **High-speed I/O**: HMCs utilize the logic die to implement high-speed signaling circuits inside memory. This realizes higher off-chip memory bandwidth between the CPU and HMCs with fewer CPU pins (e.g., 6.25x higher off-chip bandwidth with 10% fewer pins compared to DDR3 [27]).

- **Abstract memory interface**: HMCs provide a packetized abstract interface to the CPU by implementing DRAM controllers on the logic die (vault controller in Figure 1.2). Thus, CPU-side memory controllers can send high-level abstract commands (e.g., reads and writes) as packets to HMCs without taking care of the low-level characteristics of memory devices such as timing constraints, page-mode accesses, or refreshes. This simplifies the design of CPU-side memory controllers.

- **Compute-capable memory commands**: The HMC 2.0 standard [28] includes a number of compute-capable memory operations, called *atomic request commands*. By issuing such commands through the packet-based command interface, the CPU can offload simple computation (e.g., atomic add, atomic compare-and-swap, atomic bitwise operations, etc.) to HMCs.

As can be seen from above, 3D-stacked DRAM provides a cost-effective substrate to build complex functionality inside main memory. However, designing an intelligent main memory system based on it requires many more challenges to be solved. In particular, the key question that we address in this dissertation is how to design a PIM architecture that can maximize the benefit of PIM while minimizing the effort to implement and utilize such architectures. Introducing the PIM concept to existing systems requires cross-layer re-design of computer systems from hardware architectures to applications and programming models and, if such modifications involve significant effort, it can potentially prevent widespread adoption of PIM.

1.3 Dissertation Overview

The goal of this dissertation is to architect a high-performance, energy-efficient memory hierarchy by *leveraging the characteristics of emerging memory technologies*. The first part of this dissertation proposes four novel cache architectures that realize energy-efficient on-chip caches based on STT-RAM. The key objective of these approaches is to minimize the performance and energy impact of inefficient write operations of STT-RAM caches. The following summarizes the key ideas of each architecture:

- **LASIC (Chapter 3)** utilizes STT-RAM to construct an instruction cache to minimize the impact of inefficient write operations. Then, it incorporates a small SRAM cache called loop cache, which allows us to leverage the non-volatility of STT-RAM by power-gating the instruction cache inside small loops.
- **Lower-Bits Cache (Chapter 4)** filters out frequent bit-flips at lower bits of application data (commonly known as narrow bit-width characteristics) by adding a small SRAM cache that filters out lower bit changes from the main cache.
- **Prediction Hybrid Cache (Chapter 5)** designs a new hardware structure called write intensity predictor, which predicts blocks that will be frequently written during their lifetime. Then, it uses such information to guide the block placement between SRAM and STT-RAM regions in SRAM/STT-RAM hybrid caches.
- **DASCA (Chapter 6)** introduces a new concept called dead write, which is defined as a write operation that can bypass the cache without increasing the miss rate, and proposes a dead write predictor to predict/bypass dead writes to reduce the write energy consumption of STT-RAM last-level caches.

The second part of this dissertation presents four intelligent main memory systems based on logic-enabled DRAM and their host-side support. Our work explores the diverse design space of PIM architectures, from those that *maximally* modify the existing system to fully exploit the benefits of PIM to those that *minimally* modify the existing system to realize low-cost, seamless integration in the near term. The following summarizes the key ideas of each work:

- **Link Power Management for Hybrid Memory Cubes (Chapter 7)** reduces the off-chip link energy consumption of Hybrid Memory Cubes (HMCs) by (1) dynamically scaling the number of active off-chip links per HMC according to the actual link loads and (2) leveraging the existing logic die of HMCs to avoid wasting off-chip traffic in the presence of aggressive prefetching.

- **Tesseract (Chapter 8)** proposes a scalable PIM accelerator for large-scale graph processing. By integrating specialized in-order cores and a message passing communication scheme on the existing logic die of 3D-stacked DRAM and interfacing them with a domain-specific programming model, Tesseract achieves memory-capacity-proportional performance and good programmability.

- **PIM-Enabled Instructions (Chapter 9)**, or PEI, presents a low-cost PIM architecture and programming model for seamless integration into conventional systems. The key idea is to (1) expose PIM operations to programmers as host processor instructions (which we call PEIs), (2) fully support cache coherence and virtual memory without modifications to existing mechanisms, and (3) adaptively determine the location of PEI execution based on data locality.

- **Aggregation-in-Memory (Chapter 10)** proposes a holistic approach to locality-adaptive PIM systems for energy efficiency. It consists of (1) cache-conscious aggregation, which allows PIM operations to be executed at any level of the memory hierarchy according to data locality, (2) an instruction-based seamless programming model, and (3) a fully automated compilation toolchain that can transform existing applications to use PIM operations with zero programmer intervention.

Chapter 2

Previous Work

This chapter provides brief introduction to previous work on utilizing STT-RAM and logic-enabled DRAM in memory systems.

2.1 Energy-Efficient On-Chip Caches based on STT-RAM

STT-RAM caches have been considered as a promising alternative to conventional SRAM caches due to their lower static power and better scalability. Dong et al. [29, 30] modeled the performance and energy characteristics of an STT-RAM L2 cache and evaluated its impact on system performance and energy consumption. The STT-RAM cache model was later released as an open-source tool called *NVSim* [30], which has been used in most of prior works on STT-RAM cache architectures for their evaluations. These studies, along with other early-stage works [31–34], concluded that long write latency and high write energy are the most prominent drawbacks of STT-RAM caches. In this section, we review existing architectural approaches to reducing the impact of inefficient write operations of STT-RAM caches.

2.1.1 Hybrid Caches

One of the most popular approach to reducing the write energy consumption of STT-RAM caches is *hybrid caches*. Hybrid caches are motivated by the fact that different memory technologies have different pros and cons. For example, STT-RAM provides

much lower static power than SRAM, while SRAM provides much more efficient write operations than STT-RAM. Therefore, combining small SRAM caches with large STT-RAM caches can facilitate taking advantage of low dynamic energy of SRAM and low static power of STT-RAM at the same time.

Typically, SRAM/STT-RAM hybrid caches are organized by partitioning each cache set into a few SRAM ways and many STT-RAM ways in a set-associative cache [31–33, 35, 36]. For such architectures, there are two mechanisms that determine their effectiveness: (1) how to decide which cache blocks to allocate into SRAM instead of STT-RAM on a cache miss (which we call *block placement policy*) and (2) what to do if the initial placement is turned out to be suboptimal.

Sun et al. [31] presented a distributed SRAM/STT-RAM hybrid cache architecture where some of STT-RAM banks are replaced with SRAM banks. In this architecture, cache blocks loaded by read (or write) misses are allocated to STT-RAM (or SRAM) based on the intuition that blocks loaded by write misses are more likely to receive writes in the future. Since this simple policy can be suboptimal (e.g., if a cache block is loaded by a read miss, all consecutive writes to it will incur energy-consuming STT-RAM writes), they also propose to migrate STT-RAM blocks to SRAM after two successive writes to them. Li et al. [37] also used a block allocation policy based on access types of cache misses and a migration mechanism for a distributed cache architecture with SRAM and STT-RAM banks.

Wu et al. [33] proposed *read-write aware hybrid caches*, where a cache block loaded by a load (or store) miss is allocated to STT-RAM (or SRAM). In order to correct suboptimal placement, cache blocks that receive consecutive hits in the wrong region (e.g., write hits in STT-RAM or read hits in SRAM) are migrated to the opposite region. The *region-based hybrid cache architecture* proposed by Wu et al. [32, 38] has similar policies, in addition to a simple mechanism to prevent excessive migrations.

Li et al. [39] identified set-level non-uniformity of write activities and proposed to share SRAM ways in a set with neighboring sets. Since there are usually only a few SRAM ways in a SRAM/STT-RAM hybrid cache in order to minimize the static energy consumption of SRAM, sharing SRAM ways facilitates more efficient use of scare SRAM resources, thereby enabling higher write energy reduction.

Most of existing hybrid caches use migration to correct suboptimal block placement; however, frequent migration can incur noticeable energy overhead because each migration operation generates one read and one write to each region to swap cache blocks. Thus, there have been several techniques to reduce this migration overhead. Li et al. [40] proposed *compiler-assisted preferred caching*, in which migration-intensive

data is identified at compile time and is gathered in a few cache blocks that are guided to be allocated in SRAM. Li et al. [41] proposed *migration-aware compilation*, which rearranges data layout in a way that consecutive accesses to a cache block are likely to be of the same type. Both approaches target a single-level cache hierarchy (in embedded systems) as migration behavior of a multi-level cache hierarchy is hard to be estimated at compile time. Chen et al. [36] developed a compiler that provides hints about write-intensive blocks at compile time and a hardware mechanism that performs dynamic optimization considering both static information and runtime characteristics such as capacity pressure and dynamic write counts. Li et al. [42] devised *software dispatch*, which utilizes compile-time (e.g., heap data analysis) and OS-level (e.g., different virtual segments) information to guide block placement of hybrid caches. Their approach was evaluated on a system with write-through L1 caches, in which case write behavior of the L2 cache becomes more predictable.

Contrary to these existing approaches, we take a different path by providing accurate block placement at runtime, thereby eliminating the need for migration and/or recompilation of applications. Our mechanism, called *prediction hybrid cache*, will be explained in Chapter 5.

2.1.2 Volatile STT-RAM

Conceptually speaking, there is an energy barrier between '0' state and '1' state in each MTJ and write energy can be thought as the minimum energy that ensures overcoming this energy barrier to change the state of the MTJ. Therefore, the write energy of STT-RAM can be reduced by lowering this energy barrier, which can be achieved by adjusting the planar area or the thickness of the MTJ. However, lowering the energy barrier comes with a drawback that it reduces the retention time of an STT-RAM cell, i.e., stored data can be lost after a certain period of time. This is because lower energy barrier makes thermal noise easier to accidentally flip the MTJ state, which causes bit errors. Nevertheless, such *volatile* STT-RAM design can be useful for on-chip caches where the lifetime of data is relatively short (e.g., less than one second), even for large last-level caches. However, this requires cache blocks to be refreshed if their lifetime is longer than the retention time of STT-RAM cells.

Smullen et al. [43] explored the design space of such a volatile STT-RAM cache and evaluated its benefits through architectural simulation. Sun et al. [44] proposed cache hierarchy design that consists of STT-RAM with different retention levels. They also designed a hybrid cache of short-retention and long-retention STT-RAM cells (analogous to SRAM and STT-RAM, respectively, in original hybrid caches) to minimize

the refresh overhead. Jog et al. [45] suggested 10 ms as an appropriate retention time of STT-RAM cells based on their application-driven study and presented a volatile STT-RAM cache that discards unmodified cache blocks after the retention time instead of refreshing them.

It should be noted that refresh in volatile STT-RAM is different from that of DRAM. In DRAM, charges contained in each cell slowly leak over time, which is recovered by a refresh operation. Thus, performing refresh guarantees that the data will survive until the next refresh period. On the other hand, the data loss mechanism of volatile STT-RAM is different in that lowering the energy barrier of an MTJ allows lower thermal noise to flip the state, and thus, increases the probability of *stochastic* bit errors, which can happen *at any time*. This indicates that performing DRAM-style refresh (i.e., reading the data and writing it back to the same location) does not improve the reliability of data in volatile STT-RAM [46]. Instead, it should be able to detect whether bit errors have happened or not, correct them, if any, and rewriting the corrected value back to memory. This requires STT-RAM caches to have error correction code (ECC) and perform periodic scrubbing, which was not included in the aforementioned research.

2.1.3 Redundant Write Elimination

It has been known that many application programs exhibit value locality, in which many write operations do not change the original data or change only a portion of data. This happens due to many reasons. For example, the result of computation may be identical to the original value (e.g., adding zero to a value), in which case updating the original value with the new result will become a redundant write operation. Also, if the processor updates only part of an L1 cache block (which can happen because a cache block consists of multiple processor words), writing back the L1 cache block to the L2 cache will generate redundant bit writes in its unmodified portion. Eliminating such redundant writes at various granularity can reduce unnecessary write energy consumption of STT-RAM.

Lee et al. [47] proposed *partial writes*, which tracks whether the contents of each L1 cache block are actually modified or not so as to propagate only the actually modified cache blocks to the non-volatile L2 cache. Joo et al. [48] proposed *read-before-write*, which cancels write operations if they do not change the stored data. For this purpose, each write operation is preceded by a read operation, which adds a small latency overhead to each write operation. While the first two approaches were originally designed for a different non-volatile memory technology called phase-change RAM, the same mechanisms can be applied to STT-RAM caches as well. Zhou et al. [49]

presented *early write termination*, which samples the old MTJ resistance during the early stage of a write operation (i.e., no extra read latency overhead) and early terminates redundant bit writes. Sensing MTJ resistance during a write operation is possible because a typical 1T-1MTJ STT-RAM cell uses the same current path for read and write operations. Zheng et al. [50] proposed *variable-energy writes*, in which each bit-level write is terminated after the corresponding MTJ reaches the target resistance level. Such circuit-level techniques are orthogonal to architectural approaches.

2.2 Intelligent Main Memory based on Logic-Enabled DRAM

2.2.1 PIM Architectures in the 1990s

Back in the 1990s, several researchers proposed to put computation units inside memory to overcome the memory wall, including EXECUBE [16], Terasys [17], Intelligent RAM [18, 19], Active Pages [20], FlexRAM [21], and DIVA [22]. These proposals share two common characteristics: (1) fully programmable computation units (e.g., general-purpose processors, VLIW, field-programmable logic, etc.) are integrated into memory to provide flexibility and (2) computation units are built on DRAM dies to provide low-latency, high-bandwidth access to data stored in DRAM. However, not only did such approaches incur too much additional cost in DRAM manufacturing processes (which has been optimized for higher density), but also computation units on DRAM dies were much slower than those based on high-speed logic process due to the differences in manufacturing processes. Due to these reasons, the industry moved toward increasing the off-chip memory bandwidth instead of adopting the PIM concept due to costly integration of computation units inside DRAM dies.

Other than performing computation inside memory, a few prior works examined the possibility of placing prefetchers near memory. Yang and Lebeck [51] proposed to perform prefetching of linked data structures near memory. Solihin et al. [52] integrated a memory processor inside memory, which runs a user-level thread for correlation prefetching. Hughes and Adve [53] presented a near-memory prefetch engine for linked data structures.

2.2.2 Modern PIM Architectures based on 3D Stacking

With the advancement of 3D integration technologies, the PIM concept is regaining attention as it becomes more realizable [23, 54]. In this context, it is critical to examine specialized PIM systems for important domains of applications [23, 27, 55–57].

Zhu et al. [56, 57] developed a 3D-stacked PIM architecture for data-intensive workloads. In particular, they accelerated sparse matrix multiplication and mapped graph processing onto their architecture by formulating several graph algorithms using matrix operations.

Pugsley et al. [27] evaluated the PIM concept with MapReduce workloads. A MapReduce application has two distinctive phases of execution: map and reduce. The map phase is embarrassingly parallel with no communication among different workers, whereas the reduce phase is communication-intensive due to shuffling of outputs from the map phase and producing the final results based on them. In order to simplify the architecture design, they proposed to execute only the map phase inside memory while the reduce phase is executed on the host processor. Due to this, their architecture is not suitable for workloads that require frequent communication.

Zhang et al. [58] proposed to integrate GPGPUs with 3D-stacked DRAM for in-memory computing. They used a low-frequency version of commodity GPUs as in-memory processors to meet power and thermal constraints of 3D-stacked DRAM and found that using in-memory GPUs can potentially improve energy efficiency to a great extent, which however comes with performance overhead in some applications due to lower computation throughput.

Eckert et al. [59] studied the thermal feasibility of PIM in 3D-stacked DRAM. According to their analysis based on industrial-grade thermal model, they concluded that 3D-stacked DRAM with passive heat sink can sustain up to 8.5 W of power budget for in-memory computation units (which is in the level of power budget for a low-power laptop processor). This implies ample opportunities for integrating fairly complex logic into memory to realize PIM architectures.

Farmahini-Farahani et al. [60] proposed a PIM architecture based on commodity DRAM devices. The key idea is to integrate hardware accelerators (e.g., CGRAs in the paper) into standard DDR3 DIMMs by adding a logic layer to each DRAM chip and modifying data organization across different DRAM chips in a DIMM. This introduces minimal changes to DRAM dies (e.g., addition of TSVs and wider global I/O lines) and no modifications to the host processor.

Nair et al. [61] proposed *active memory cube (AMC)*, a processing-in-memory system targeted for scientific computing. AMC integrates vector processors on the logic die of an HMC and interfaces them with the host processor via off-chip links of HMCs. It also provides a set of APIs to control the data placement in AMCs and compiler support for offloading parallel loops annotated by OpenMP directives. According to their evaluations, AMC achieves much higher power efficiency than other supercomputer

systems for common kernels used in scientific computing, DGEMM (dense general matrix multiplication) and DAXPY (dense $aX + y$ computation). Sura et al. [62] later showed how to efficiently utilize AMC with a better programming model and compiler support.

Akin et al. [63] presented a data reorganization accelerator based on 3D-stacked DRAM. It adds small additional logic on the logic die of 3D-stacked DRAM to implement permutation-based data layout reorganization. The proposed accelerator was evaluated for two use cases: (1) software-transparent data layout optimization, which periodically remaps application data in a way to increase parallelism and row buffer locality, and (2) acceleration of data reorganization routines, which are popular in scientific computation (e.g., matrix transpose).

Several studies have proposed 3D-stacked system designs targeting memory-intensive workloads. Kgil et al. [64] proposed the *PicoServer* architecture, where multiple DRAM dies are integrated with a logic die containing many in-order cores for higher energy efficiency with a smaller form factor. Loh [65] proposed to stack DRAM dies on top of server processors and modify the memory architecture to fully utilize high bandwidth and parallelism provided by 3D-stacked DRAM. Ranganathan [66] introduced a concept called *nanostores*, where each chip integrates dense non-volatile memory and power-efficient compute cores with 3D stacking and multiple chips are networked together with on-board connectors to scale the system. Gutierrez et al. [67] presented *Mercury* and *Iridium*, a 3D-stacked server design that combines many in-order processors, a 10 GbE network interface controller (NIC), and memory (either DRAM or Flash) together in a package to efficiently service key-value store workloads. Such *standalone* 3D-stacked systems can be broadly classified as PIM, with one difference that they do not have the host processor to interface with. This sometimes leads to different design choices in architecture design, which will be demonstrated in Chapter 9 and 10.

2.2.3 Modern PIM Architectures on Memory Dies

Some previous works tried to exploit *existing internal structures of memory devices* and add minimal extra logic to implement simple computation in memory. Such approaches are different from PIM architectures in the 1990s in that the former minimizes modifications to memory dies, whereas the latter diminished due to excessive cost overheads of integrating *complex* logic circuits (e.g., general-purpose processors) on DRAM dies. Some of such efforts leverage new non-volatile memory technologies (e.g., STT-RAM, phase-change RAM, resistive RAM) for PIM implementation since,

unlike DRAM, they use resistance to represent stored information, which sometimes makes it easier to implement simple computation.

Guo et al. [68] presented *AC-DIMM*, an associate compute engine in memory. AC-DIMM utilizes a modified STT-RAM cell structure to build a memory array that can be configured as RAM, content-addressable memory (CAM), or ternary CAM (TCAM) at runtime. In addition to its search capability, it also incorporates a small programmable microcontroller at the center of four memory arrays and a compute-capable reduction tree, which allow us to perform in-memory computation with data stored in memory arrays.

Seshadri et al. [55] proposed to implement bulk data copy and initialization in DRAM. Many applications and operating systems can benefit from fast copy and/or initialization of bulk data, including process forking, data structure initialization, secure memory deallocation, and so on. In order to accelerate such common operations, their mechanism, called *RowClone*, leverage internal structures of DRAM that are designed to transfer data between DRAM cells and the host system, including row buffers and internal data buses. This facilitates an entire DRAM row to be copied to another row or initialized to zero without using off-chip channels, thereby achieving higher performance and lower energy consumption.

Wang et al. [69] proposed *ProPRAM*, a low-cost implementation of simple PIM operations based on non-volatile memory. The key observation behind this work is that non-volatile memory already integrates simple logic circuits for optimizing their operations. For example, many non-volatile memory products implement data comparison write (similar to read-before-write introduced in the previous section) to reduce the amount of writes, which adds a comparator to each memory bank. Flip-N-Write [70] is another popular mechanism that inverts the data to be written if that reduces the number of bit flips, which requires an inverter and an adder to be added to each bank. ProPRAM utilizes such auxiliary logic circuits to perform simple computation in memory, such as addition, scan search, and bitwise logical operations.

Chapter 3

Loop-Aware Sleepy Instruction Cache

In the first part of this dissertation, we propose architectural techniques to design energy-efficient on-chip caches based on STT-RAM. Since the drawback of STT-RAM is in its inefficient write operations, we first explore the application of STT-RAM to caches that experience relatively infrequent write operations, i.e., instruction caches. Contrary to data caches, instruction caches show relatively infrequent write activities because data is written to the caches only on cache misses. Thus, applying STT-RAM to instruction caches can minimize the impact of its inefficient write operations while exploiting the benefits of STT-RAM.

When an instruction cache is constructed with STT-RAM, we observe that the major source of energy consumption is from its static energy consumption (e.g., 63% on average according to our simulation results). Considering that STT-RAM cells are generally considered as not consuming any leakage power (because the access transistor in each cell is turned on only for cells involved in read or write operations), this static energy consumption is from the peripheral circuits of STT-RAM. The reason why the peripheral circuits of STT-RAM instruction caches consume high static power is its tight performance constraint. In contrast with L2 caches using LP (Low Power) cells, L1 instruction caches are typically constructed with HP (High Performance) cells to achieve low access latency at a cost of high static power. Due to this, the static power

This chapter is originally published in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2014 [71].

of an L1 instruction cache is sometimes comparable or even higher than that of an L2 cache. For example, according to our simulation results based on CACTI [13] and NVSim [14], a 16 KB STT-RAM L1 instruction cache with HP cells consumes almost the same static power as that of a 512 KB SRAM L2 cache with LP cells. Therefore, it is very important to reduce the static energy consumption of instruction caches despite their relatively small size.

In order to mitigate the impact of this problem, we focus on an unexplored opportunity that STT-RAM caches are very good targets to apply the power-gating technique. Traditionally, SRAM caches suffer from a trade-off between state retention and leakage power reduction [72]. Non-state-retentive techniques usually offer a much better reduction in leakage power than state-retentive ones, but extra cache misses due to state loss incur performance degradation and additional dynamic energy consumption. Moreover, state-retentive power gating on SRAM cells needs careful adjustment to retention voltage due to process variation. However, STT-RAM caches do not have such issues because of its non-volatility, and thus, power-gated STT-RAM caches can achieve significant leakage power reduction and negligible performance loss at the same time.

In this chapter, we describe our new architecture for power-gated STT-RAM instruction caches, called *Loop-Aware Sleepy Instruction Cache (LASIC)*. Our goal is to design an architecture and a policy that can detect long idle time of STT-RAM instruction caches in advance and therefore can greatly reduce energy consumption by power-gating the caches with minimal impact on performance.

3.1 Architecture

Figure 3.1 shows a high-level overview of our architecture. We use STT-RAM only for the instruction cache and leave the data cache hierarchy unmodified. Our architecture is inspired by the observation of an instruction cache behavior in program loops. While the processor executes a loop, the instruction cache repeatedly serves a small number of instructions within the loop body until its termination. Therefore, by adding a tiny buffer called a *loop cache*, instructions for the loop body can be retrieved from the buffer instead of the L1 instruction cache as long as the buffer can contain the entire loop body. This enables an opportunity to turn off the L1 instruction cache during the execution of loops smaller than the capacity of the loop cache, which takes a significant portion of the program execution (e.g., programs spend 34% of their execution time on average to execute small loops according to our simulation results). In the following subsections,

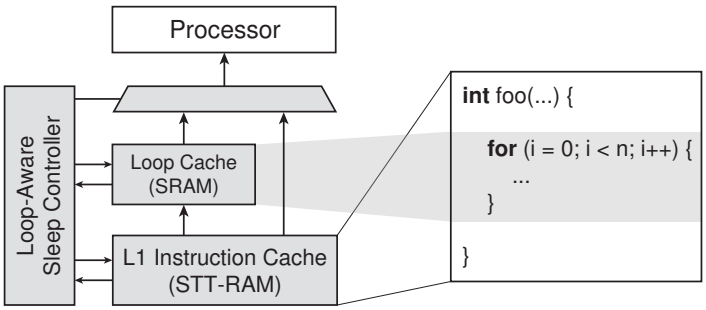


Figure 3.1: Proposed STT-RAM instruction cache architecture.

we describe the details of our architecture, including the loop cache architecture and our strategy to control the loop cache and the power-gated instruction cache.

3.1.1 Loop Cache

A loop cache is a tiny buffer that stores instructions inside a small loop body [73]. When a small loop that can fit into the loop cache is detected by monitoring short backward branches[1] (called *trigger branches*), instructions are loaded into the loop cache and then the instructions for the next iterations are fetched from the loop cache until the processor exits the loop. Trigger branch detection is performed by a simple additional hardware unit; neither the processor nor target applications need to be modified at all. Also, a loop cache itself typically does not have a tag array or valid bits because it is used only for loops whose body does not exceed its capacity (i.e., no conflict miss). Instead, a counter is added to determine whether the loop is completely loaded into the loop cache or not (once a backward branch is detected, the counter is used to count up the fetched instructions to see if all the instructions in the loop are cached). It is known that a loop cache has the ability to reduce the dynamic energy consumption of an L1 instruction cache.

The loop cache architecture proposed by Lee et al. [73] has a drawback that the loop cache cannot be used for loops that have internal control flows (e.g., loops containing conditional statements, nested loops, etc.). In their scheme, the loop cache is deactivated at any control flow that is not caused by the trigger branch instruction. This is because, for loops with internal control flows, the counter-based controller is not sufficient to determine whether a specific line of a loop cache is valid or not. Sato and Sato [74] added

[1]Hereafter, we use the word "branch" only for denoting a direct jump. The reason why we deal with only direct jumps is that indirect jumps require the controller to be able to access the register file of the processor to calculate their target addresses.

sfb bits to detect forward branches, but it still does not support loops with complicated control flow including nested loops. In order to overcome this drawback, Gordon-Ross et al. [75] proposed pre-loaded loop caches, in which target loops are determined at design time based on profiling results. Although their technique can further reduce the number of L1 cache accesses, it limits the number of loops that can be cached in a loop cache since all of the selected loops are loaded at the time of program initialization.

In order to overcome such limitations, we remove the counter from the loop cache and add a valid bit array and boundary address registers that store the start and the end address of a loop. Also, each entry of the loop cache is modified to hold one block (composed of a few instructions) rather than one instruction. The block size of the loop cache is set to that of the L1 instruction cache.

In our architecture, small loop detection is performed as follows. When a backward branch (a direct jump with a negative displacement) is detected, the size of a loop formed by the branch and its start address are calculated by the displacement field of the branch instruction. If the loop is smaller than the capacity of the loop cache, the loop cache is activated and the start/end addresses of the loop are recorded to the boundary address registers.

Since the loop cache does not have tags, it needs a special mechanism to check for a hit. When the loop cache is accessed, the controller checks whether the requested address is located inside the loop by comparing it with the boundary address registers. If so, the valid bit corresponding to the address is read to check for a hit and the block data is loaded from the L1 instruction cache on a miss. On the other hand, when the requested address is outside of the loop, the loop cache is deactivated and all valid bits are set to zero. This can be done within a cycle because they are implemented with registers, which can be updated in parallel. Note that the valid bit array incurs negligible area cost and power consumption because loop caches are very small. For example, a 2 KB loop cache with 64-byte blocks needs only 32 registers to store valid bits.

At each context switch, every block in the loop cache is flushed to prevent the aliasing problem. Note that this would not harm the effectiveness of loop caches since context switching occurs infrequently (10 to 100 ms in typical operating systems).

3.1.2 Loop-Aware Sleep Controller

We propose a loop-aware sleep controller to coordinate the loop cache and the power-gated L1 instruction cache. Figure 3.2 is a state diagram of the proposed controller.[2]

[2]Self-loop edges are omitted for each state but it is implicitly used when none of the transition conditions are satisfied. "≤ LC" denotes that a loop is smaller than or equal to the loop cache capacity.

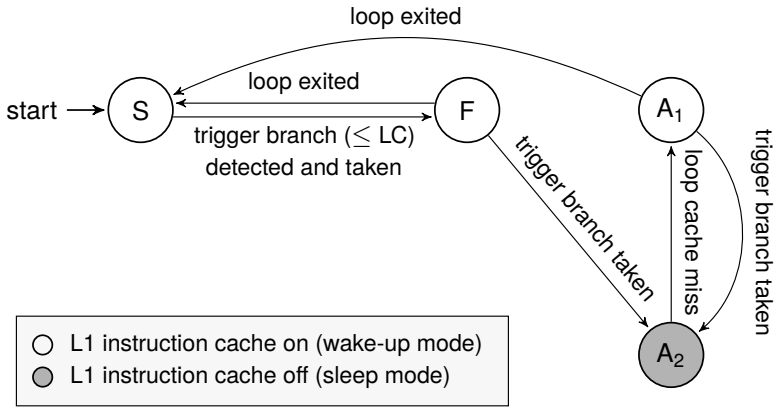


Figure 3.2: State diagram of the loop-aware sleep controller.

Our policy is composed of three states related to the loop cache, which are *standby* (S), *fill* (F), and *active* (A_1 and A_2). The *active* state is divided into two substates according to whether the L1 instruction cache is in sleep mode (A_2) or not (A_1).

In the *standby* state, the loop cache is not used at all and instructions are fetched from the L1 instruction cache. The controller monitors trigger branches and changes its state to *fill* with the mechanism explained in Section 3.1.1.

In the *fill* state, the L1 instruction cache still serves instructions to the processor, but every fetched instruction is also written to the loop cache. The controller still monitors trigger branches and changes its state to *active* when the trigger branch is taken again, indicating that the loop cache is ready to be used.

In the *active* state, the loop cache serves instructions to the processor instead of the L1 instruction cache, and thus, the L1 instruction cache can be turned off completely (A_2). On a loop cache miss, the controller powers up the L1 instruction cache to load the corresponding block from it (A_1). This situation could occur when the loop contains control-dependent code that has not been executed yet in previous iterations. At the end of the iteration, the instruction cache is turned off again to reduce the static energy consumption. Although this policy might turn the L1 instruction cache on and off several times per loop, such a case would not occur frequently[3] because, as long as a block has at least one control-independent instruction, it will be loaded into the loop cache during the *fill* state (i.e., no loop cache misses during the *active* state).

Among the three loop cache states, the *active* state consumes the lowest power. In this state, not only is static power consumption greatly reduced through powered-

[3]According to evaluation results, the average miss rate of the loop cache in the active state was 0.5%.

Table 3.1: Simulator Configuration

Component	Configuration
Processor	1 GHz, in-order, two-issue superscalar
Loop Cache (SRAM)	2 KB, direct-mapped, 64 b blocks
L1 I-Cache (SRAM or STT-RAM)	16 KB, 4-way, 64 B blocks, 1/1-cycle (SRAM read/write) or 1/11-cycle (STT-RAM read/write) latency
L1 D-Cache (SRAM)	16 KB, 4-way, 64 B blocks, 1-cycle latency
L2 Cache	512 KB, 8-way, 64 B blocks, 9-cycle latency
Main Memory	200-cycle latency

off instruction caches, but dynamic power consumption is also reduced because the per-access energy of the loop cache is lower than that of the L1 instruction cache. On the contrary, the *fill* state consumes slightly higher dynamic power than the *standby* state due to the write energy of the loop cache. However, its effect on the total power consumption is small in general because it lasts for only one iteration per loop.

3.2 Evaluation and Discussion

3.2.1 Simulation Environment

We evaluate the effectiveness of our architecture through architectural simulation. We use a modified version of SimpleScalar 3.0 [76] with the processor and memory hierarchy configuration shown in Table 3.1. Our cache hierarchy is designed to resemble that of ARM Cortex A8. We set the size of the loop cache to 2 KB based on our experiments (see Section 3.2.4 for sensitivity analysis on the capacity of the loop cache).

Simulations are performed by running SPEC CPU2000 [77] benchmark applications until termination. Since the reference inputs are too large to be used for simulation, we use the *lgred* sets from MinneSPEC [78], a carefully reduced version of SPEC CPU2000 datasets for simulations.

We use CACTI 6.5 [13] and NVSim [14] to model the characteristics of SRAM and STT-RAM caches, respectively, under 45 nm technology. Table 3.2 shows the energy consumption of the loop cache and the instruction caches based on SRAM and STT-RAM. Note that STT-RAM is still an emerging technology, and thus, the modeling results could be different from those of other research. Although our STT-RAM caches are a bit ahead of the current technology, we believe that the technology will become

Table 3.2: Characteristics of the Instruction Caches

	Loop Cache	SRAM L1 I-Cache	STT-RAM L1 I-Cache
Read Energy (pJ)	7.49	20.1	33.0
Write Energy (pJ)	7.49	20.1	21.6
Static Power (mW)	2.43	21.6	10.4

available in the near future. STT-RAM has already been used as L1 caches with read latency less than 1 ns in several previous works [34, 43, 79].

Power gating of the STT-RAM caches is modeled by referring to the results from the previous research that applied power gating to peripheral circuits of SRAM caches [80]. This is because the only source of static power in STT-RAM is peripheral circuits, which have a similar structure to those of SRAM. Among the three sleep modes defined in their research, the deepest sleep mode (the biggest leakage power reduction and the longest wake-up latency) is selected for our experiments. This is because our technique does not incur much performance degradation even with long wake-up latency. According to their data, we set the static power of the STT-RAM instruction caches in the sleep mode and wake-up latency to 0.73 mW and three cycles, respectively.

The area and power overhead of the loop-aware sleep controller are estimated by synthesizing our Verilog implementation of the controller with Synopsys Design Compiler and TSMC 45 nm technology library. According to the synthesis results, the controller occupies less than 5% of the area of the instruction cache. Also, its power consumption is only 0.5 mW to 1 mW depending on its state, which is negligible compared to that of the instruction cache.

3.2.2 Energy

Figure 3.3 shows the comparison in energy consumption of the instruction cache under various architectures. We compare our architecture (LASIC) against the following four different configurations:

- BASE-SRAM/-STT: SRAM/STT-RAM baseline
- LOOP-SRAM/-STT: SRAM/STT-RAM instruction cache with a loop cache but without power gating

As can be seen in the figure, the vanilla STT-RAM instruction cache does not provide any improvement over the SRAM cache. On average, the STT-RAM baseline consumes almost the same amount of energy compared to the SRAM baseline. Although

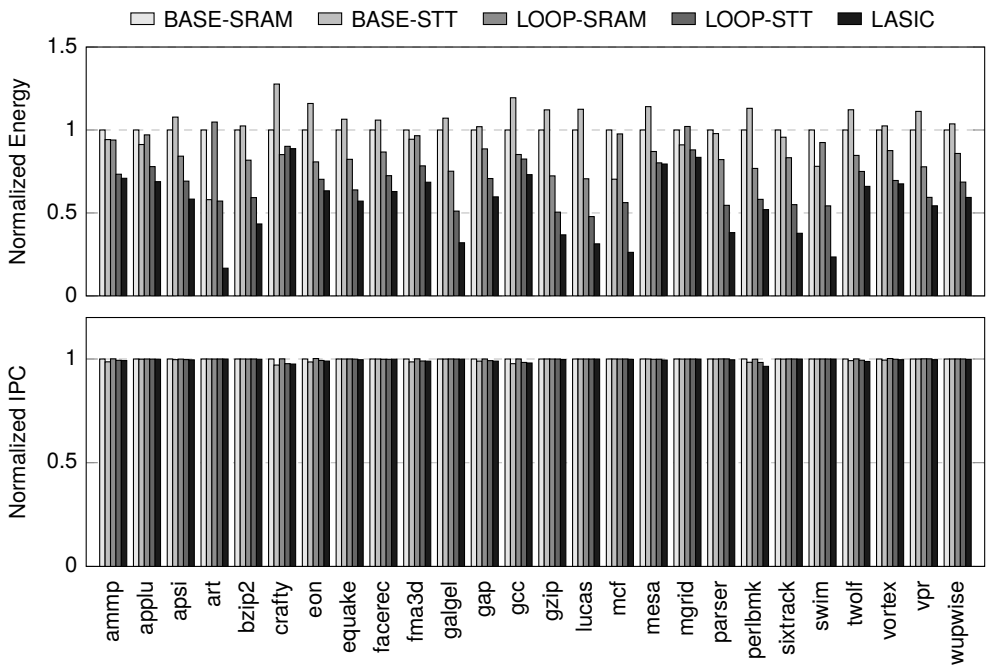


Figure 3.3: Comparison in the energy consumption (above) and the performance (below). The results are normalized to the SRAM baseline.

the STT-RAM cache reduces the static energy consumption by 52%, its 64% higher read energy offsets the benefit of static energy reduction. Note that the impact of high write energy is rather minimal as write energy takes less than 1% of dynamic energy consumption. This is an expected result considering that instructions are written to the instruction cache only on misses (for block fill) and the miss rate itself is very low (0.3% on average).

However, the loop cache is able to mitigate the effect of increased read energy by reducing read operations to the L1 instruction cache. Due to this, the instruction cache starts to take advantage of low static power of STT-RAM, i.e., LOOP-STT consumes 24% lower energy than LOOP-SRAM. At the same time, the major source of energy consumption shifts from dynamic energy to static power of the L1 instruction cache (41% on average). This indicates the need for techniques that can reduce the static energy consumption of STT-RAM instruction caches.

Compared to LOOP-STT, LASIC further reduces the energy consumption by 23%. This comes from a 50% reduction in static energy consumption through power gating of the L1 instruction cache. As a result, LASIC consumes 49% and 50% lower

Table 3.3: Summary of the Experimental Results

	BASE-STT	LOOP-SRAM	LOOP-STT	LASIC
Static Energy	−52%	−11%	−41%	−62%
Dynamic Energy	+73%	−48%	−29%	−30%
Total Energy	+0.5%	−14%	−34%	−49%

energy compared to BASE-SRAM and BASE-STT, respectively, thereby facilitating an energy-efficient application of STT-RAM into instruction caches.

Table 3.3 summarizes the experimental results. Each number shows the difference from the SRAM baseline and negative percentages indicate energy reduction.

3.2.3 Performance

Figure 3.3 also shows the performance comparison between LASIC and conventional architectures. Compared to the SRAM baseline, the STT-RAM baseline shows negligible performance degradation. Considering that using STT-RAM for the L1 data cache can incur noticeable performance overhead [79], we can conclude that instruction caches are a favorable target for STT-RAM. As mentioned previously, the reason why long write latency does not take effect is that instruction caches exhibit infrequent write operations.

Also, LASIC shows almost identical performance compared to the STT-RAM baseline. Theoretically, it is possible that our architecture might incur performance overhead compared to the LOOP-STT because, each time the L1 instruction cache is accessed during its sleep state (e.g., after loop cache deactivation or on misses), the processor has to wait until it is turned on. However, according to our experiments, such performance overhead is limited to only 0.24% on average because our policy has a tendency to turn off the instruction cache only when long sleep duration is expected. Once the instruction cache is turned off, its sleep lasts for approximately two hundred cycles on average, which is far longer than the break-even time of power gating.

3.2.4 Sensitivity Analysis

Figure 3.4 shows the energy consumption of LOOP-STT and LASIC under different loop cache capacity. The energy consumption is normalized to that of the STT-RAM baseline. According to the experimental results, larger loop caches achieve higher reductions in the static energy consumption of the L1 instruction cache because they enable bigger loops to take advantage of our technique. However, larger loop caches are not always

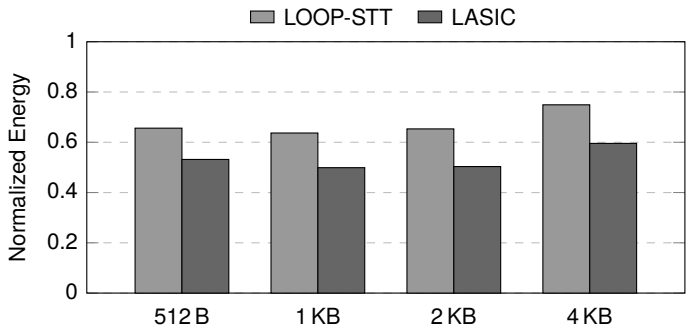


Figure 3.4: Sensitivity analysis on the capacity of the loop cache.

beneficial because they dissipate higher static and dynamic power. Among 0.5 KB to 4 KB loop caches, the 2 KB loop cache is the one that balances between these two aspects, which is why we we select it as the default configuration for our experiments.

We also measured the performance overhead under various wake-up latencies from 4 cycles to 30 cycles. According to our results, the performance overhead is limited to only 3% on average even with the longest wake-up latency.

3.3 Summary

In this chapter, we introduced a novel architectural technique to reduce the static energy consumption of STT-RAM instruction caches by exploiting abundant instruction locality in program loops. Our architecture incorporates an improved loop cache that supports any kind of small loops and controls the sleep mode of the power-gated L1 instruction cache based on it. Evaluation results show that our architecture achieves a 50% energy reduction in the L1 instruction cache with less than 0.1% performance overhead compared to the STT-RAM baseline. Based on the results, we can conclude that reducing static energy consumption is still a very important problem even for STT-RAM caches, and our architecture can effectively alleviate the energy implication of this problem.

Chapter 4

Lower-Bits Cache

As explained in Section 1.2.1, inefficient write operations are the most important drawback of STT-RAM caches. For STT-RAM data caches, the easiest way to reduce its impact is the read-before-write strategy [48], in which data is read before every write operation to cancel the write request if the data to be written is exactly the same as the existing one. Although this strategy can filter some of write requests, it is not very usual that the entire data in a cache block is not modified (consider an L2 cache where a write request involves an entire L1 cache block instead of just one word).

In this chapter, we present a simple cache architecture that reduces the write energy consumption of STT-RAM L2 caches under the read-before-write strategy. To make read-before-write more effective, our architecture exploits the observation that upper bits are not updated as frequently as lower bits. Our architecture hides such frequent bit-flips at lower bits from the STT-RAM cache, which allows the read-before-write strategy to cancel more write requests.

4.1 Architecture

In many applications, computed values vary within small ranges, and thus, most of write requests do not change upper bits of those values. Our architecture leverages

This chapter is originally published in Proceedings of the International Symposium on Circuits and Systems, 2012 [81].

this observation by incorporating a small SRAM cache, called *lower-bits cache*, that hides frequent bit changes at lower bits from the STT-RAM cache. For this purpose, the lower-bits cache holds the lower half of every word in cache blocks written by the L1 data cache. Not only can it coalesce several write requests as in conventional multi-level caches, but also it makes frequent value changes hidden from the L2 cache so that the read-before-write strategy can effectively cancel write requests.

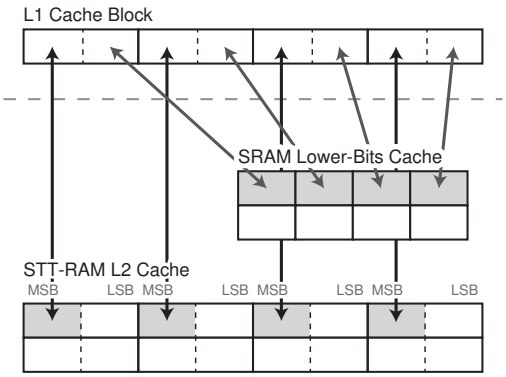
Figure 4.1 shows a detailed diagram of operations that distinguish our architecture from conventional caches. In this figure, each word in an L1 data cache block is divided into upper bits and lower bits (denoted by broken lines). All events are handled in the same way as conventional caches, except for the following four events:

Writeback/Read from the L1 Cache. When the L1 data cache writes its dirty cache block back to the L2 cache, lower bits are separated from the cache block and stored in the lower-bits cache (Figure 4.1a). The upper bits are written to the L2 cache only if the data to be written is actually different from what is stored in the L2 cache, according to the read-before-write strategy.

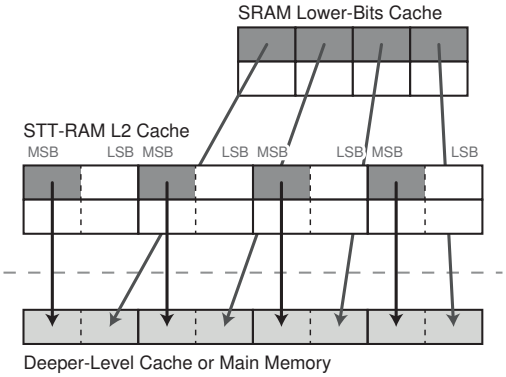
In this architecture, the lower-bits cache is accessed in parallel with L2 cache access for every read operation. When the access hits in the lower-bits cache, the data to be provided for the L1 cache is a combination of the upper bits from the L2 cache and the lower bits from the lower-bits cache (Figure 4.1a). This is necessary for correctness since lower bits of some L2 cache blocks may not be up-to-date if the most recently written data is stored only in the lower-bits cache.

Eviction from the L2 Cache. When a dirty L2 cache block is evicted, both the L2 cache and the lower-bits cache are checked in parallel (Figure 4.1b). If the lower-bits cache has the corresponding block, writeback is performed by combining upper bits of the L2 cache block and lower bits of the lower-bits cache block. After the writeback, both the L2 cache block and the lower-bits cache block are invalidated. If the lower-bits cache does not contain the corresponding block, the access is handled in the same way as conventional caches.

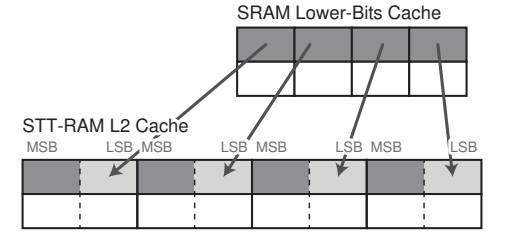
Eviction from the Lower-Bits Cache. In our architecture, the capacity of the lower-bits cache is configured to be much smaller than the L2 cache to minimize its static power consumption, and thus, some of its blocks may be evicted before the eviction of the corresponding L2 cache block (Figure 4.1c). In this case, evicted blocks are written back to the L2 cache. However, it does not occur very often as the lower-bits cache holds only the blocks written by the L1 data cache rather than all blocks in the L2 cache.



(a) Writeback/Read from the L1 data cache



(b) Eviction from the L2 cache



(c) Eviction from the lower-bits cache

Figure 4.1: Operations of the lower-bits cache.

Table 4.1: Characteristics of the L2 Cache and the Lower-Bits Cache

	SRAM L2	SRAM Lower-Bits			
	2 MB	8 KB	16 KB	32 KB	64 KB
Read Latency (ns)	3.75	0.97	1.00	1.45	2.06
Write Latency (ns)	11.63	0.97	1.00	1.45	2.06
Read Energy (pJ)	51.3	2.59	3.53	5.06	8.21
Write Energy (pJ)	1453	1.92	2.26	3.25	4.88
Leakage Power (mW)	8.93	0.19	0.34	0.71	1.39

Advantages. Our architecture provides two opportunities for write energy saving. First, separating upper bits and lower bits allows the read-after-write strategy to cancel a substantial portion of write requests since only upper bits, which are infrequently changed, are written to the L2 cache on every write operation. Second, if the lower-bits cache can hold its block until the corresponding L2 cache block is evicted, lower bits of the block do not need to be written to the L2 cache at all.

4.2 Experiments

4.2.1 Simulator and Cache Model

We use our modified version of SimpleScalar 3.0 [76] for experiments. Our configuration includes a 2.5 GHz four-issue out-of-order processor and a memory hierarchy that resembles that of Intel Core 2 architecture, which has 32 KB, 8-way set-associative L1 instruction and data caches with 64-byte blocks, and a 2 MB, 16-way set-associative L2 unified cache composed of 16 STT-RAM banks with 64-byte blocks. We use 8, 16, 32, or 64 KB of 16-way SRAM caches with 32-byte blocks as the lower-bits cache.

We model the power consumption and the access latency of caches by using a modified version of CACTI 5.1 [82]. We refer to the previous work [49] for STT-RAM cell parameters for 45 nm technology. We use Low Operating Power (LOP) peripherals for the peripheral circuits of the L2 cache and the cells/peripherals of the lower-bits cache. Table 4.1 summarizes the modeling results.

We use 12 integer benchmarks from SPEC CPU2000 [77] as simulation workloads. Since reference inputs are not suitable for simulation due to the long execution time, all experiments are performed with *lgred* datasets from MinneSPEC [78], a reduced version of SPEC CPU2000 datasets for simulation.

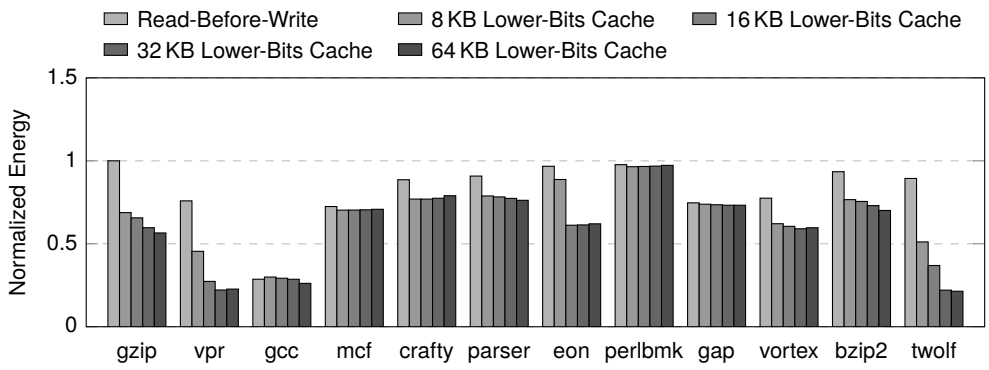


Figure 4.2: Comparison of dynamic energy consumption between the baseline and the lower-bits cache (normalized to the STT-RAM baseline).

4.2.2 Results

We compare the dynamic energy consumption of our architecture with the STT-RAM baseline with/without the read-before-write strategy. As shown in Figure 4.2, our architecture with a 32 KB lower-bits cache reduces 46% of the dynamic energy consumed by the L2 STT-RAM cache on average.

Importantly, our architecture outperforms the STT-RAM baseline with the read-before-write strategy. This implies that upper bits of each word in a cache block are not modified at all in many cases. According to our results, compared to the STT-RAM baseline with the read-before-write strategy, a 32 KB lower-bits cache reduces the number of bytes written to the STT-RAM L2 cache by 16% to 88%.

We also observe that larger lower-bits caches achieve higher dynamic energy reductions. This is because larger lower-bits caches are helpful in avoiding the eviction of lower-bits cache blocks, which reduces the writes of lower bits to the STT-RAM L2 cache. However, enlarging the lower-bits cache is not always beneficial since it increases the static and dynamic power consumption of the lower-bits cache itself.

Figure 4.3 shows the system performance (instructions per cycle) and the energy consumption of the L2 cache and the lower-bits cache (if any). Under the optimal lower-bits cache capacity of 32 KB, our architecture achieves 25% energy reduction on average, as expected from the reduction in dynamic energy consumption. However, in some cases, our architecture increases the energy consumption by up to 6% because (1) the read-before-write strategy requires one additional read for every write, which incurs additional energy overhead if only a few of write requests are cancelled by it, and (2) the lower-bits cache itself increases the static energy consumption.

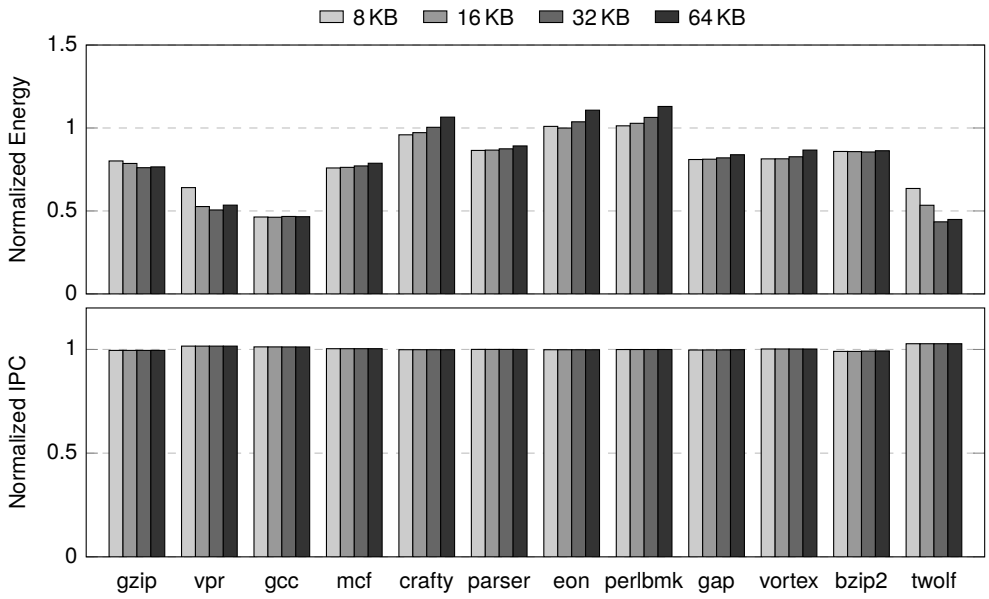


Figure 4.3: Energy consumption (top) and IPC (bottom) of the lower-bits cache (normalized to the STT-RAM baseline).

Our architecture also improves the system performance by 0.4% on average (up to 2.8%). Although the read-before-write strategy sometimes degrades the performance due to its extra read operation prior to each write operation, our architecture offsets such overhead by canceling a larger portion of write requests.

4.3 Summary

In this chapter, we proposed the lower-bits cache to reduce the dynamic energy consumption of STT-RAM caches. It is based on the observation that bit changes are skewed towards lower bits of each word, indicating that upper bits do not change frequently. In order to hide frequent changes at lower bits, our cache architecture incorporates a small SRAM cache, called lower-bits cache, which stores only the lower half of every word in cache blocks. Experimental results show that the lower-bits cache with the read-before-write strategy reduces 46% of dynamic energy consumption (or 25% of total energy) of the L2 cache compared to the STT-RAM baseline.

Chapter 5

Prediction Hybrid Cache

In the previous chapter, we introduced an architectural technique to reduce the write energy consumption of STT-RAM caches by leveraging value similarity. However, such an approach would be effective only if the applications exhibit value locality, which might not be the case in some applications. For example, workloads that heavily use floating-point values have relatively low value locality due to the high bit-flip entropy of floating-point values.

A more general way to alleviate the effect of inefficient write operations is to utilize both SRAM and STT-RAM together in a cache, called SRAM/STT-RAM hybrid caches [31–33, 35, 37–39]. In hybrid caches, cache ways are partitioned into SRAM and STT-RAM regions and frequently written cache blocks, which we call *write-intensive* blocks, are allocated to the SRAM region (instead of the STT-RAM region) to reduce the write activity to the STT-RAM region. Since the write energy of SRAM is much smaller than that of STT-RAM, it helps to reduce the amount of write operations to STT-RAM, thereby reducing dynamic energy consumption.

The energy efficiency of hybrid cache architectures highly depends on how accurately we can identify write-intensive blocks. If too many blocks are chosen to be allocated to the SRAM region, frequent evictions from the SRAM region will degrade the hit rate and thus will hurt overall performance. On the other hand, if some write-intensive

This chapter is originally published in Proceedings of the International Symposium on Low Power Electronics and Design, 2013 [83] and IEEE Transactions on Computers, 2016 [84].

blocks are allocated to the STT-RAM region, large energy penalty would be incurred due to high write energy of STT-RAM.

Despite such importance of this problem, solutions that have been proposed by previous work are either too simple or too static. More specifically, many researchers proposed to determine the region to place a block based only on the type of operations (read/write) [31, 33, 37, 39], which is not good enough to accurately identify write-intensive blocks. Recent studies proposed compilation techniques to identify write-intensive data at compilation time and to provide the information as a hint to the runtime system [36, 40, 42]. One drawback of these approaches is that applications need to be recompiled for each architecture, which is not possible in many cases. Moreover, it is inherently impossible for compiler-based approaches to take dynamic characteristics of applications into account. Although they showed the efficiency of their techniques on single-level cache hierarchy [40] or write-through caches [42], their static nature makes them very hard to be adopted into writeback caches since the number of writebacks on each block is difficult to estimate at compilation time. This could be a serious limitation for STT-RAM caches, which prefer the writeback policy to maximize write coalescing at upper levels of the memory hierarchy.[1]

In this chapter, we propose a novel mechanism, called *write intensity predictor*, to dynamically predict the write intensity of each cache block. It is based on the key observation that there is a high correlation between write intensity of blocks and addresses of load/store instructions that incur misses of the blocks. The predictor exploits this observation by keeping track of instructions that tend to load write-intensive blocks and utilizing that information to predict write intensity of blocks that will be accessed in the future. This enables dynamic prediction of write intensity even for cache blocks accessed for the first time, which is not possible with simple per-block counters.

Based on this concept, we propose a new hybrid cache architecture, called *prediction hybrid cache*. In this architecture, block placement is determined by prediction information from the write intensity predictor. Moreover, it periodically monitors application characteristics and dynamically adjusts the threshold of the write intensity predictor for better adaptivity.

[1]Note that write-through caches request too many writes to deeper-level caches thereby increasing write energy significantly.

5.1 Problem and Motivation

5.1.1 Problem Definition

The objective of this work is to predict write intensity of cache blocks on their misses. To solve this problem, we first introduce a new cost model for each block of hybrid caches. The idea is to formulate energy cost of placing a block into STT-RAM instead of SRAM. Formally, the *cost* of a block is defined as

$$\begin{aligned} c &= N_{\mathrm{r}}\Delta E_{\mathrm{r}} + N_{\mathrm{w}}\Delta E_{\mathrm{w}} \\ &= N_{\mathrm{r}} \times \left(E_{\mathrm{r}}^{\mathrm{STT}} - E_{\mathrm{r}}^{\mathrm{S}}\right) + N_{\mathrm{w}} \times \left(E_{\mathrm{w}}^{\mathrm{STT}} - E_{\mathrm{w}}^{\mathrm{S}}\right) \end{aligned} \tag{5.1}$$

where N_{r} and N_{w} denote the number of reads and writes to the block until its eviction, $E_{\mathrm{r}}^{\mathrm{STT}}$ and $E_{\mathrm{w}}^{\mathrm{STT}}$ denote normalized read and write energy of the STT-RAM region, and $E_{\mathrm{r}}^{\mathrm{S}}$ and $E_{\mathrm{w}}^{\mathrm{S}}$ denote those of the SRAM region.[2] Our cost model includes only dynamic energy consumption since static energy consumption is dependent on system performance, which will be optimized by a separate mechanism (see Section 5.3.4).

Based on this model, we define *write-intensive* blocks as cache blocks whose costs are higher than or equal to κ where κ is the write intensity threshold. Note that, since the cost of a block is determined by the number of reads and writes *until the time of its eviction*, it is impossible to calculate the write intensity of a block *on its miss* (i.e., when the block is first loaded into the cache). In addition, κ needs to be chosen carefully so as to allocate neither too many nor too few blocks into the SRAM region. This work tackles these two challenges through architectural solutions.

5.1.2 Motivation

Our key idea is that write intensity of blocks can be inferred from the load/store instructions that load the blocks into the cache. In other words, there is a strong correlation between costs of blocks and the instructions that incur cache misses of the blocks. We call such instructions *trigger instructions* and focus on their potential to be a good proxy for *predicting* write intensity of blocks.

Figure 5.1 shows an experimental evidence of this intuition obtained from four SPEC CPU2006 benchmark applications under a two-level cache hierarchy (details of the simulation configuration are given in Section 5.4.1). In this figure, the x-axis shows

[2]For SRAM/STT-RAM hybrid caches, ΔE_{r} is usually negative since higher density of STT-RAM makes interconnects shorter and thus reduces energy dissipated from wires [85, 86], while ΔE_{w} is positive. However, our formulation is not limited to such a specific case and can model other memory technologies with different ΔE_{r} and ΔE_{w}.

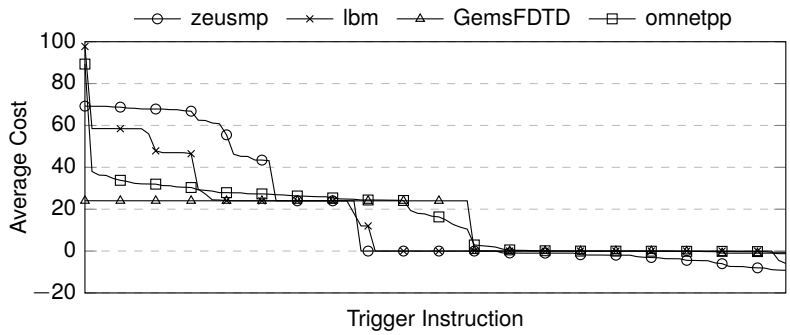


Figure 5.1: Distribution of the average cost per block according to trigger instructions in four applications. Markers are drawn every five points for better visibility.

the trigger instructions sorted by their y-values, while the y-axis shows the average cost per L2 cache block loaded by each of the trigger instructions. We set ΔE_{r} and ΔE_{w} of our cost model to -1 and 24, respectively. For better visibility, the figure shows only the top 100 trigger instructions in terms of the number of cache blocks loaded. Although not shown in the figure, we confirmed that other applications in the benchmark share similar trends with the ones shown here.

The most important observation from the figure is that blocks loaded by some trigger instructions tend to have much higher average costs than others. This indicates that blocks that will be loaded by such trigger instructions in the future are also expected to have high costs, which in turn implies good predictability of write intensity. Therefore, one can predict write intensity of a block *on its miss* by observing whether the corresponding trigger instruction has already loaded write-intensive (or high-cost) blocks or not. Our architecture is designed on the basis of this idea to accurately predict write intensity of blocks. Although a few studies observed a correlation between cache hit/miss information and the instruction addresses that cause the cache accesses [87–90], our work is the first to discover and exploit the correlation between write intensity of blocks and instructions that induce their cache misses.

5.2 Write Intensity Predictor

Based on the motivation, we formulate the problem of identifying write-intensive blocks as finding trigger instructions that tend to load write-intensive (or high-cost) blocks, which we call *hot* trigger instructions.[3] For this purpose, we propose a new hardware

[3]We use the term *cold* trigger instructions as the opposite meaning.

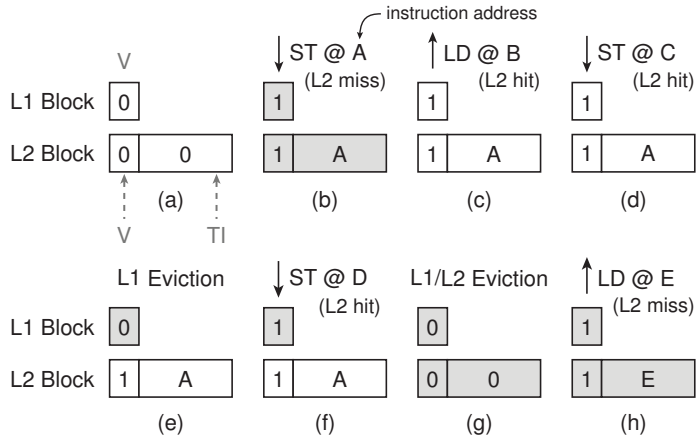


Figure 5.2: An example of updating TI fields (shaded boxes indicate that the values are updated).

structure called *write intensity predictor*, which is comprised of two steps: keeping track of trigger instructions and identifying hot trigger instructions. In the following subsections, we describe the details of the two steps. For this, we assume without loss of generality that write intensity of L2 cache blocks needs to be predicted under a two-level writeback cache hierarchy. This mechanism could also be used for other variants of cache hierarchies, such as multi-level cache hierarchies or write-through caches.

5.2.1 Keeping Track of Trigger Instructions

The first step is to keep track of trigger instruction information for each block. To achieve this goal, we add an extra field TI (trigger instruction) to each L2 cache block to store an address of a trigger instruction. When an L1 cache miss occurs, the address of the instruction that incurs the miss is sent to the L2 cache along with the block fill request. This information is recorded into the L2 cache block only if the request also incurs an L2 cache miss. Through this, we can keep track of the trigger instruction address of each block until the L2 cache block is evicted.

Figure 5.2 illustrates an example sequence of updating TI fields. The figure shows changes in V (valid) and TI of an L2 cache block and the corresponding L1 cache block according to the sequence of load/store instructions. Let us assume that both blocks are initially invalid (a). After executing the store instruction having address A, the TI field of the L2 cache block is updated to A as it loads the block into the L2 cache (b). Since the block is now valid, subsequent load/store instructions do not update its TI

field (c and d). Its value remains the same even on L1 cache misses as long as the block is still valid in the L2 cache (e and f). The TI field of the L2 cache block is cleaned on its eviction (g) and the next access to it updates the TI field with the new address E (h).

Note that trigger instruction addresses do not need to be exact since inaccurate write intensity prediction does not affect the semantics of program execution at all. In this work, the lower-order 12 bits[4] of a trigger instruction address are stored into a TI field instead of the entire address in order to reduce the storage overhead.

5.2.2 Identifying Hot Trigger Instructions

The second step is to identify hot trigger instructions, and eventually, to predict write intensity of blocks. The basic idea is to keep track of per-block costs and associate that information with the corresponding trigger instructions to predict write intensity of blocks to be loaded in the future.

First, we maintain the cost per L2 cache block so that we can determine write intensity of a block on its eviction. An 8-bit cost field is added into each L2 cache block for this purpose. The cost field of a block is incremented by ΔE_{r} and ΔE_{w} on each read and write operation to the block, respectively, and is reset on its eviction.

Based on this information, we update the states associated with trigger instructions to detect hot trigger instructions. For this, we add a lookup table called *predictor table*, which maintains a state for each trigger instruction. Each state is represented by a two-bit saturating counter where values 0 and 1 indicate cold trigger instructions and values 2 and 3 indicate hot trigger instructions. When a block is evicted, the value of the cost field is compared with the write intensity threshold κ. If the cost reaches or exceeds the threshold, the state of the corresponding trigger instruction is updated toward 'hot' state (i.e., incremented); otherwise, it is updated toward 'cold' state (i.e., decremented).

We found that this mechanism is very similar to a branch predictor, a well-known component of computer systems. The idea of transforming our problem into branch prediction is simple and intuitive. If the cost of a cache block is higher (lower) than the write intensity threshold, we consider it as a taken (untaken) branch at the corresponding trigger instruction. Through this formulation, taken and untaken branches in branch prediction are translated respectively into hot and cold trigger instructions in our problem.

[4] In fact, the length of a trigger instruction address is determined by the size of the predictor table (see Section 5.2.2) since trigger instruction addresses are used only for accessing the predictor table, which is tag-less and direct-mapped. Since our architecture uses a predictor table with 4096 ($= 2^{12}$) entries (see Section 5.4.1), lower-order 12 bits are enough to index predictor table entries.

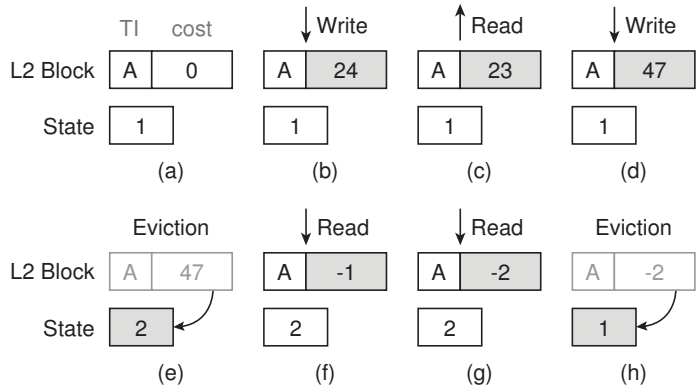


Figure 5.3: An example of updating per-block costs and states ($\Delta E_r = -1$, $\Delta E_w = 24$, $\kappa = 20$).

To help better understanding of the proposed scheme, an example of the operating sequence is provided in Figure 5.3. It shows an L2 cache block and a predictor table entry (state) corresponding to the trigger instruction of the block. For simplicity, we assume ΔE_r, ΔE_w, and κ to be -1, 24, and 20, respectively, in this example. Initially, the cost is set to zero and the state is set to weakly cold (i.e., value 1) (a). The cost is incremented by 24 for each write operation (b and d), whereas it is decremented by one for each read operation (c). Since the cost of the evicted block is higher than the threshold, the state of the corresponding trigger instruction is updated toward 'hot' state (e). On the contrary, assuming that the block is loaded again and is read twice before eviction (f and g), the state of the corresponding trigger instruction is updated toward 'cold' state on its eviction since its cost after the two read operations is lower than the threshold (h).

5.2.3 Dynamic Set Sampling

Our prediction scheme requires two additional fields to be added into each L2 cache block: the TI and cost fields. Although they introduce relatively small storage overhead (i.e., 24 bits per block), it might not be negligible for large caches.

Therefore, we introduce the concept of dynamic set sampling [91] into the write intensity predictor to reduce the area overhead. The key idea behind this is that characteristics of the entire cache (in this case, whether a trigger instruction is hot or cold) can be generalized by observing a few cache sets. This is accomplished by adding a small partial tag array called *sampler* that contains only a few cache sets and simulates cache replacement behavior of those sets by sampling the cache accesses to them.

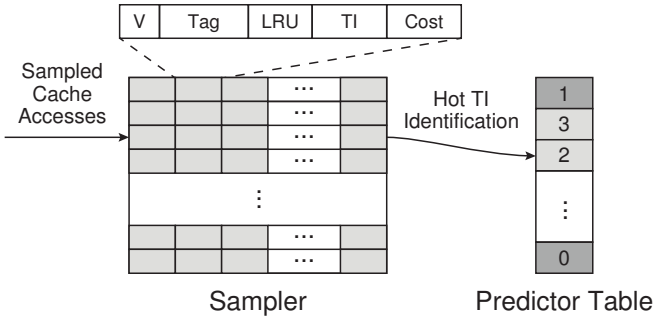


Figure 5.4: The write intensity predictor (not to scale).

In this work, the sampler is constructed by selecting $1/32$ of the entire sets[5] based on the simple-static policy [91]. Each sampler entry contains a valid bit, a 16-bit partial tag,[6] replacement policy bits (i.e., LRU bits), a 12-bit TI field, and an 8-bit cost field. On cache accesses to the sampled sets, the TI field and the cost field of the corresponding sampler entry are updated according to the aforementioned scheme. The sampler follows the plain LRU replacement policy (whereas the L2 cache uses a modified version of the LRU replacement policy for hybrid caches) in order to prevent write intensity prediction from being affected by previous block placement decisions.

Adopting dynamic set sampling in our architecture reduces the storage overhead incurred by the TI field and the cost field down to 1.28 bits per block without any noticeable impact on prediction accuracy, which will be discussed more in Section 5.5.5.

5.2.4 Summary

Figure 5.4 summarizes the proposed write intensity predictor design. It consists of a sampler (structurally similar to tag arrays) and a predictor table (similar to bimodal branch predictors). The sampler simulates cache replacement behavior of sampled sets by monitoring cache accesses to them. When a sampler entry is evicted, the predictor table entry addressed by the value of the TI field is updated accordingly. This allows us to keep track of hot trigger instructions in the predictor table.

[5]This sampling ratio has been used by many previous works based on dynamic set sampling [88,91,92].

[6]In the sampler, only the lower-order 16 bits of tags are stored because infrequent mismatches on sampler tags have negligible impact on prediction.

5.3 Prediction Hybrid Cache

In this section, we present a new hybrid cache architecture called *prediction hybrid cache* based on the proposed write intensity predictor.

5.3.1 Need for Write Intensity Prediction

As introduced before, the energy efficiency of hybrid caches heavily depends on the policy of selecting blocks to be allocated to the SRAM region instead of the STT-RAM region. One common approach to this is to allocate blocks that are loaded due to write (or store) misses[7] into the SRAM region and other blocks to the STT-RAM region [31, 33, 37, 39]. The major weakness of such an approach is that it has a high possibility to miss many write-intensive blocks due to its simplicity. For example, it cannot detect the case where a block is loaded by a read (or load) miss but is written many times after the miss. To compensate the weakness, the approach is usually used along with block migrations, which swap a block in the STT-RAM region that has been written frequently with another block in the SRAM region. However, migrations could incur significant energy overhead since each migration requires at least one read and write operation for each region as studied in previous work [93].

Our inspiration is that, if we could know in advance whether a block will be write-intensive or not, the efficiency of hybrid caches can be improved by simply allocating predicted write-intensive blocks into the SRAM region. This information is exactly what the write intensity predictor provides. Therefore, we utilize the write intensity predictor to improve block placement in hybrid caches for better energy efficiency.

5.3.2 Organization

Figure 5.5 shows the overview of the prediction hybrid cache. The system is composed of an SRAM/STT-RAM hybrid cache and a write intensity predictor. Similarly to other hybrid cache architectures, each cache set is partitioned into SRAM ways and STT-RAM ways. For illustration purposes, we assume a six-way set-associative cache with two SRAM ways and four STT-RAM ways.

Unlike the data arrays, the tag array is constructed only with SRAM since each tag contains status bits and replacement information, which can be updated frequently [85, 94]. The sampler and the predictor table are also constructed with SRAM due to their small size. The write intensity predictor can optionally include a *threshold selector*, a

[7] We use the term 'load/store miss' as a cache miss incurred by a load/store instruction, respectively, as in the previous work [33].

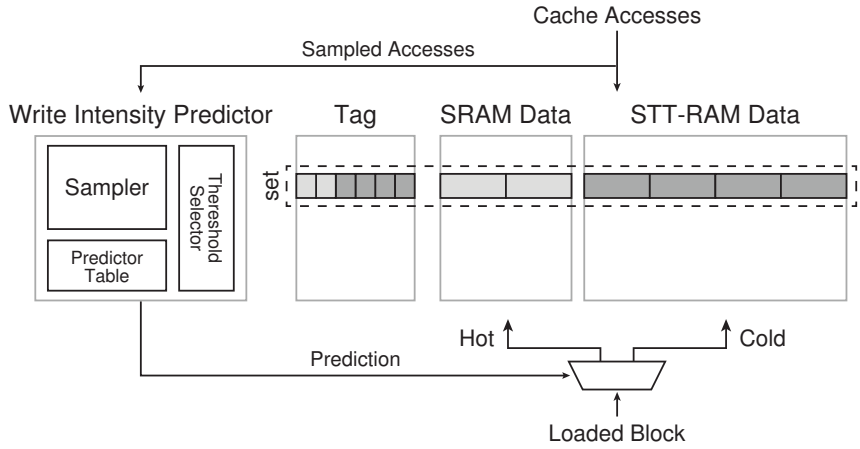


Figure 5.5: Overview of the prediction hybrid cache.

hardware structure that adaptively adjusts the write intensity threshold according to application characteristics (see Section 5.3.4 for details).

5.3.3 Operations

Cache accesses in prediction hybrid caches are performed as in conventional caches. Only two small modifications are introduced to support write intensity prediction. First, accesses to the sampled sets are also sent to the sampler of the write intensity predictor in order to update necessary information for identifying hot trigger instructions. Second, the lower-level caches (e.g., L1 cache) are modified to transfer the lower-order 12 bits of the trigger instruction address along with the read request [87–90]. If there are multiple levels of caches between processors and prediction hybrid caches, such information is simply propagated through the intermediate caches.

The uniqueness of our architecture is in its block placement policy. On a cache miss, the predictor table entry indexed by the corresponding trigger instruction address is checked to see if it is in 'hot' state. If so, the loaded block is allocated into the SRAM region since blocks loaded by hot trigger instructions are expected to be write-intensive; otherwise, it is allocated into the STT-RAM region.[8] After that, a victim block is selected among blocks in the target region according to the LRU replacement policy.

[8]For non-blocking caches, miss status holding registers (MSHRs) need to keep track of trigger instruction addresses associated with in-flight misses. Otherwise, that information will not be available when the block data arrives from the next level of memory.

Unlike previous approaches, our architecture does not migrate blocks between the SRAM region and the STT-RAM region. The rationale behind this is that the migration scheme requires per-block counters to track the number of writes for each block and extra control logic and buffers to swap blocks between two regions [31–33, 35, 37–39], which incur nonnegligible overhead in terms of both energy and area. In particular, such overhead is not worthwhile considering that migration plays a role of correcting imperfect block placement, which takes only a small portion in our scheme due to the high prediction accuracy (see Section 5.5.3 for accuracy analysis). According to our experimental results, incorporating migration into our scheme further reduces the energy consumption of the cache by 6.3% on average.[9] We can achieve such an improvement since the migration technique is orthogonal to our work. However, to clearly show the sole impact of our contribution, we do not consider migration throughout this work.

5.3.4 Dynamic Threshold Adjustment

One of the most important parameters of our architecture is the write intensity threshold κ. If the threshold is set either too high or too low, the miss rate increases thereby increasing the energy consumption of the main memory because most of the blocks are allocated into one region, which leaves the other underutilized. Between the two cases, too low thresholds are more critical because of the small capacity of the SRAM region. However, too high thresholds also incur high write energy consumption due to the increased number of blocks allocated to the STT-RAM region. Such behavior is well illustrated in Figure 5.6, which shows the trend of write energy consumption and miss rates under various thresholds. The former is write energy consumed by the L2 cache during the execution of zeusmp, which is normalized to that of $\kappa = 0$. The latter is shown as a difference from the miss rates of the iso-capacity STT-RAM-only cache with the LRU replacement policy (e.g., +1% of miss rate difference indicates that the L2 cache miss rate increases by 1% compared to a conventional cache that uses the LRU replacement policy).

The difficulty here is that the optimal threshold depends on the distribution of per-block costs, which varies across different applications and program phases even within a single application. The problem of determining the best threshold becomes even more challenging in multicore systems since it is nearly impossible to profile every possible combination of workloads that could be run simultaneously.

[9] This result was obtained by exploring the best threshold for each single-core application under our architecture with migration and comparing it with Static PHC (see Section 5.5 for its meaning).

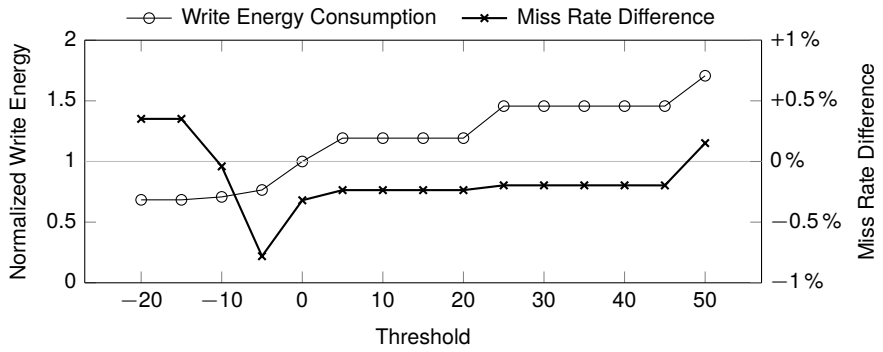


Figure 5.6: Impact of the write intensity threshold on write energy consumption and miss rates in zeusmp.

Therefore, we propose a hardware-based approach to adapt the write intensity threshold in runtime by monitoring application characteristics. It is designed to find the smallest threshold among the ones that do not increase the miss rate compared to the baseline replacement policy (e.g., LRU replacement policy in this work). More specifically, our mechanism incrementally finds the best threshold by either incrementing or decrementing the current threshold according to the application characteristics. This raises two follow-up questions: (1) When should the threshold be incremented/ decremented? (2) How much should it be changed?

For the first question, we estimate miss rates under different replacement policies as follows. In order to reduce the overhead of estimation, we use the number of misses in sampled sets (see Section 5.2.3) during a fixed interval as a proxy for miss rates of the entire cache.

- M_{LRU}: the number of misses under the LRU replacement policy.
- M_0: the number of misses under the hybrid cache replacement policy with the current threshold κ_0.
- M_+: M_0 with a higher threshold $\kappa_+ > \kappa_0$.
- M_-: M_0 with a lower threshold $\kappa_- < \kappa_0$.

M_0 can be obtained by monitoring misses in sampled sets of the hybrid cache itself because it uses the current threshold κ_0 for block placement. For this purpose, we add a 16-bit miss counter, which is incremented on each cache miss in the sampled sets.

On the other hand, M_{LRU} assumes a conventional cache with the LRU replacement policy and thus may not be obtained from the hybrid cache. Fortunately, however, it can be estimated from the sampler without any additional hardware because the

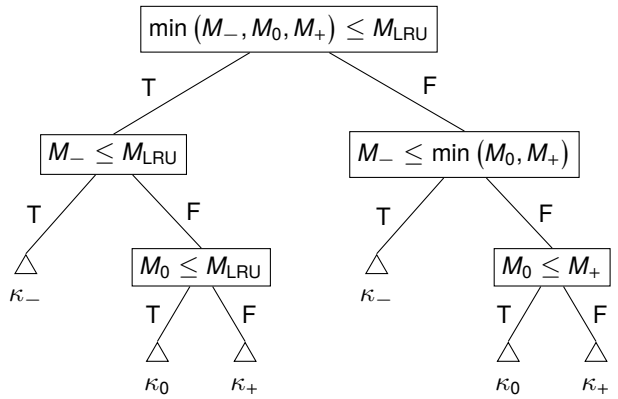


Figure 5.7: Decision tree for dynamic threshold adjustment.

sampler follows the LRU replacement policy when replacing its entries (explained in Section 5.2.3). Thus, we add a 16-bit miss counter, which is incremented every time a sampler entry is installed (i.e., sampler miss).

In the case of M_+ and M_-, we add a predictor table and a partial tag array for each of them to simulate cache replacement under κ_+ and κ_-. The predictor tables are exactly the same as that in the write intensity predictor, except that they are updated with threshold κ_+ and κ_- (instead of κ_0) using the same mechanism described in Section 5.2.2. The partial tag arrays simulate cache replacement of sampled sets under the higher and the lower thresholds by using the newly added predictor tables. For this purpose, each entry of the partial tag arrays is composed of a valid bit, a 16-bit partial tag, and replacement policy bits.

Based on these values, the write intensity threshold is adjusted periodically with the policy depicted in Figure 5.7. If any of κ_-, κ_0, or κ_+ shows the same or lower miss rate than that of the LRU replacement policy, the algorithm chooses the smallest threshold among the ones that satisfy the condition to reduce write energy consumption as much as possible (left subtree). Otherwise, it selects the one with the smallest miss count to reduce the miss rate (right subtree). After that, all miss counts are reset for the next interval.

The remaining issue is to determine the amount of threshold adjustment, i.e., the values of κ_+ and κ_-. A naive way is to set κ_+ and κ_- to $\kappa_0 + 1$ and $\kappa_0 - 1$, respectively. However, according to our experiments, it takes a long time to find the best threshold because of the wide range of cost values as shown in Figure 5.1. Simply increasing the

step size does not work either as the miss rate becomes sensitive to threshold changes in a certain range of thresholds (e.g., $-15 \leq \kappa \leq 5$ in Figure 5.6).

Therefore, we propose to monitor costs of evicted sampler entries during each interval and track the highest below-threshold cost and the lowest above-threshold cost among them for threshold candidates of κ_- and κ_+. The former and the latter can be defined as follows:

$$c_-(\kappa_n) = \begin{cases} \max C_- & \text{if } C_- = \{c \in C \mid c < \kappa_n\} \neq \varnothing \\ \kappa_n & \text{otherwise} \end{cases} \tag{5.2}$$

$$c_+(\kappa_n) = \begin{cases} \min C_+ & \text{if } C_+ = \{c \in C \mid c > \kappa_n\} \neq \varnothing \\ \kappa_n & \text{otherwise} \end{cases} \tag{5.3}$$

where C is a set of costs of evicted sampler entries during the current interval and κ_n is the write intensity threshold for the next interval determined by the decision tree in Figure 5.7. However, since it is unknown which of κ_-, κ_0, and κ_+ will become κ_n until the end of the current interval, we track c_- and c_+ for each of κ_-, κ_0, and κ_+. Note that, since $c_-(\kappa_n)$ (or $c_+(\kappa_n)$) is the maximum (or minimum) cost among the ones below (or above) the threshold κ_n, it can be tracked by adding only a single 8-bit register (which has the same bit width as the cost field in the sampler) without storing the entire C. Thus, tracking c_- and c_+ for each of κ_-, κ_0, and κ_+ requires six 8-bit registers. At the end of each interval, κ_- and κ_+ are updated to $c_-(\kappa_n)$ and $c_+(\kappa_n)$, respectively. This has an effect of enlarging the step size of threshold adjustment when the cost distribution is sparse thereby allowing faster adaptation. For example, in Figure 5.6, the threshold 20 can directly be decremented to 0 without sweeping through any intermediate values, while it is adjusted more carefully between 0 and -15. In total, our threshold selector introduces low storage overhead of 1.7 bits per block under our single-core configuration (see Section 5.4.1 for details).

5.4 Evaluation Methodology

5.4.1 Simulator Configuration

We evaluate our architecture under single-/quad-core systems by using a cycle-accurate x86-64 simulator based on Pin [95]. The simulator models 3 GHz, out-of-order, four-issue superscalar cores with 128-entry instruction window. The cache hierarchy is composed of 32 KB, 4/8-way set-associative private L1 instruction/data caches and a 1 MB per core, 16-way set-associative shared L2 cache, whose block size is 64 bytes.

Table 5.1: Characteristics of the L2 Cache

	Single-Core (1 MB)			Quad-Core (4 MB)		
	SRAM	STT	Hybrid	SRAM	STT	Hybrid
Read Latency†	10	10	10	13	13	13
Write Latency†	10	37	10/37*	13	40	13/40*
Read Energy (nJ)	0.09	0.07	0.09/0.07*	0.18	0.08	0.18/0.08*
Write Energy (nJ)	0.09	0.64	0.09/0.64*	0.18	0.69	0.18/0.69*
Static Power (mW)	14.7	2.32	5.41	69.5	7.80	20.8
Area (mm^2)	3.77	0.95	1.66	16.9	2.74	6.28

† Latencies shown in cycles under 3 GHz.
* Parameters for accesses to the SRAM/STT-RAM region, respectively.

Both the L1 data cache and the L2 cache are non-blocking caches with eight MSHRs. Cache coherence is managed by the MESI protocol without enforcing the inclusion property. The main memory consists of two dual in-line memory modules (DIMMs), each of which is comprised of one rank containing eight 2 Gb DDR3-1600 11-11-11-28 devices [96]. The memory controller uses the FR-FCFS scheduling under the open-row policy [97] and has a 64-entry request queue.

The proposed hybrid cache architecture is applied to the L2 cache and is compared against Read-Write Aware Hybrid Caches (RWHCA) [33]. The hybrid caches are configured to have 4 SRAM ways and 12 STT-RAM ways. In RWHCA, migrations are triggered on four consecutive hits in the wrong region as in the original paper. In our scheme, dynamic threshold adjustment is triggered every five million cycles (chosen based on simulation results).

Table 5.1 shows energy and delay characteristics of the L2 cache at a 45 nm technology modeled by CACTI 6.5 [13] and NVSim [14]. LOP devices are used for peripheral circuits and SRAM cells as they are known to be best matched to the characteristics of last-level caches in commercial high-performance processors [98]. STT-RAM cell parameters are obtained from the previous works [15, 99]. As shown in the table, read and write operations in hybrid caches are modeled to have different latencies and energy characteristics depending on the target region. Based on these values, we set $\Delta E_{\mathrm{r}} = -1$ and $\Delta E_{\mathrm{w}} = 24$ for the single-core system and $\Delta E_{\mathrm{r}} = -1$ and $\Delta E_{\mathrm{w}} = 6$ for the quad-core system.

We also model energy consumption of additional circuits for hybrid cache management by using CACTI 6.5. In RWHCA, a two-bit counter is added into each block for

migration, which incurs 4 KB overhead in the single-core system. In our architecture, a write intensity predictor is composed of a sampler (2.6 KB), a 4096-entry bimodal predictor table (1 KB), and a threshold selector (3.3 KB) thereby adding 6.9 KB in the single-core system. Although we use a small predictor table, our simulation results confirm that larger predictor tables do not have any noticeable impact on prediction accuracy (e.g., less than 0.1% difference in accuracy with a four times larger predictor table). The overhead of the two-bit counters, the sampler, and tag arrays of the threshold selector is quadrupled in the quad-core system due to the four times larger cache capacity, while the size of the predictor table remains the same, i.e., 19.4 KB in total. In summary, our scheme introduces 0.67%/0.45% storage overhead in the single-/quad-core system.

According the latency modeling from CACTI 6.5, the write intensity predictor is not on the critical path of cache accesses, and thus, does not increase cache access latency. This is because (1) the sampler is updated in parallel with cache accesses and (2) the predictor table is accessed only on cache misses.

Lastly, energy consumption of the main memory is modeled by Micron System Power Calculator [100].

5.4.2 Workloads

We use 16 write-intensive[10] benchmarks from SPEC CPU2006 [101] as single-core workloads. For benchmarks having more than one input set, we choose the representative ones by referring to the previous work [102]. Each benchmark is run for one billion instructions of its representative phase [103].

From the single-core workloads, we construct 12 multiprogrammed workloads as shown in Table 5.2. To evaluate with a wide range of write intensity, we first classify single-core workloads into top eight (denoted as High) and bottom eight (denoted as Low) in terms of write intensity and assemble four Highs into high1 to high4, two Highs and two Lows into mid1 to mid4, and four Lows into low1 to low4. Each workload is run for four billion instructions.

We also use four multithreaded workloads from PARSEC 2.1 [104] for multicore evaluations. Each benchmark is run for two billion instructions from the beginning of Regions-of-Interest (ROI).

[10]Write intensity of a benchmark is defined as WBPKI (writeback per kilo instruction) of the L1 data cache under the single-core system.

Table 5.2: Workloads from SPEC CPU2006

Workload	WBPKI
lbm	43.8
mcf	26.9
soplex	20.8
libquantum	18.9
leslie3d	10.8
omnetpp	10.8
GemsFDTD	10.6
gcc	8.9
h264ref	8.0
milc	7.6
zeusmp	7.3
xalancbmk	5.6
bzip2	5.2
tonto	4.6
wrf	4.4
hmmer	2.9

Mix	Workloads
high1	lbm, mcf, soplex, libquantum
high2	leslie3d, omnetpp, GemsFDTD, gcc
high3	lbm, soplex, leslie3d, GemsFDTD
high4	mcf, libquantum, omnetpp, gcc
mid1	lbm, mcf, hmmer, wrf
mid2	soplex, libquantum, tonto, bzip2
mid3	leslie3d, omnetpp, xalancbmk, zeusmp
mid4	GemsFDTD, gcc, milc, h264ref
low1	hmmer, wrf, tonto, bzip2
low2	xalancbmk, zeusmp, milc, h264ref
low3	hmmer, tonto, xalancbmk, milc
low4	wrf, bzip2, zeusmp, h264ref

5.5 Single-Core Evaluations

In this section, we compare our architecture (denoted as Dynamic PHC) against SRAM baseline, STT-RAM baseline, and Read-Write Aware Hybrid Caches (RWHCA) [33]. We also evaluate a static version of our architecture (denoted as Static PHC) as a limitation study, in which the write intensity threshold is statically selected on a per-application basis through profiling from $\kappa = -20$ to 50 at intervals of 5. For each application, we choose the best threshold that minimizes energy consumption of the L2 cache while not increasing the energy consumption of the main memory compared to the STT-RAM baseline.

5.5.1 Energy Consumption and Speedup

Figure 5.8 shows the energy consumption and speedup of different configurations under the single-core system. The energy consumption of the L2 cache includes that of additional circuits for hybrid cache management. The last set of bars labeled by 'GMEAN' represents geometric mean of energy consumption or speedup.

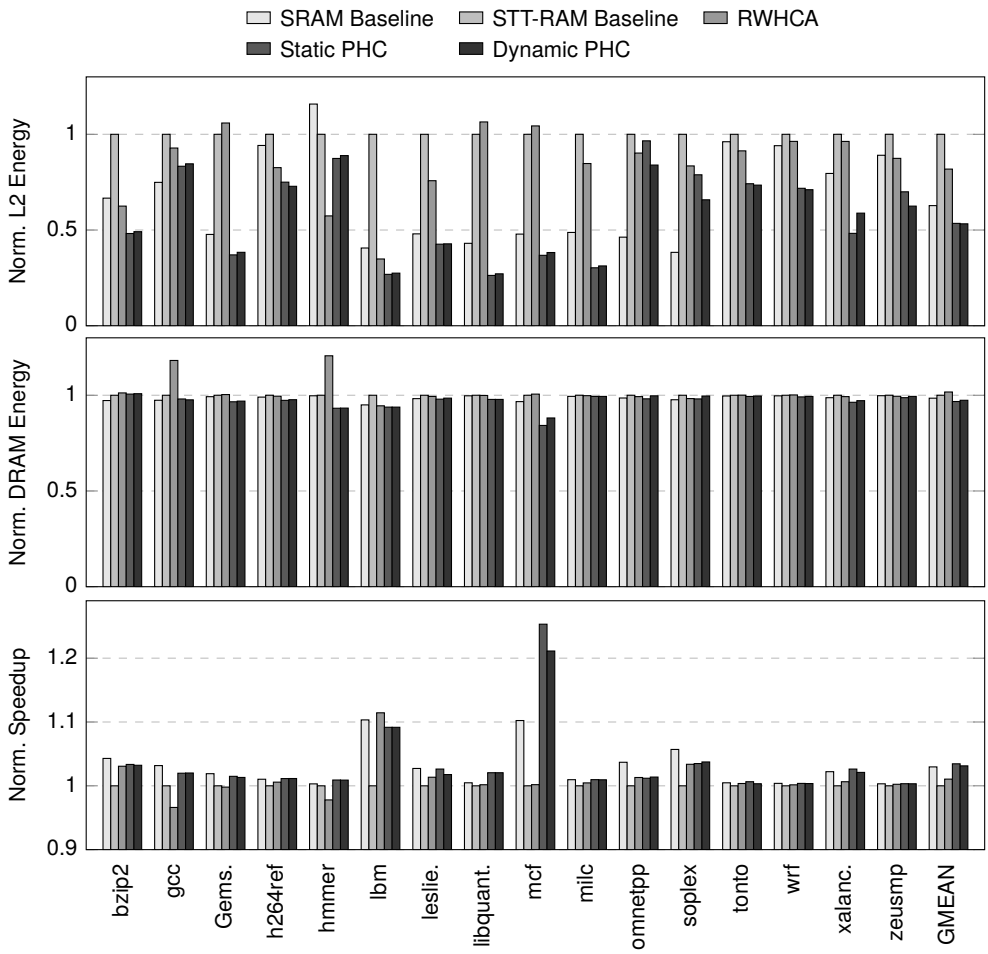


Figure 5.8: Energy consumption and speedup under the single-core system (normalized to the STT-RAM baseline).

As shown in the figure, Dynamic PHC reduces the energy consumption of the L2 cache by 47% compared to the STT-RAM baseline. Moreover, its energy consumption is even slightly less than that of the SRAM baseline. This is a promising result considering that the 16 benchmarks evaluated here are write-intensive, in which STT-RAM caches are undesirable due to its seven times higher write energy (see Section 5.5.6 for results for other benchmarks).

Compared to RWHCA, our architecture achieves a 35% reduction in the energy consumption of the L2 cache. This is because our scheme identifies more write-intensive blocks than RWHCA does in general, and thus, more write operations are handled in the SRAM region. Note that allocating more blocks into the SRAM region is beneficial as long as it does not increase energy consumption of the main memory. For example, RWHCA consumes 35% less L2 cache energy in hmmer compared to our scheme, but increases the energy consumption of the main memory by 29%, which leads to a 29% increase in total energy consumption. On average, our scheme even reduces the energy consumption of the main memory by 4.3% compared to RWHCA, indicating that our scheme achieves a better balance between write energy reduction and miss rates.

Also, Dynamic PHC reaches approximately the same level of energy efficiency as that of Static PHC without the need for per-application profiling. In some cases (e.g., omnetpp, soplex, and zeusmp), Dynamic PHC even outperforms Static PHC because it can adapt to different phases within an application at runtime. On average, Dynamic PHC consumes 0.4% less energy in the L2 cache with a 0.7% increase in energy consumption of the main memory compared to Static PHC.

Lastly, our scheme improves system performance by 3% and 2% compared to the STT-RAM baseline and RWHCA, respectively, thereby reaching the performance of the SRAM baseline. The major contribution of this speedup comes from the reduction in write operations to the STT-RAM region, each of which takes three times longer than that in the SRAM region. Also, the reduction in main memory activities also improves the performance in some benchmarks (e.g., mcf).

5.5.2 Energy Breakdown

Figure 5.9 shows energy breakdown of the L2 cache under the baselines and our architecture (Dynamic PHC). Each bar is decomposed into the static energy consumption of the L2 cache, the dynamic energy consumption of the L2 cache (further decomposed into tag access, cache read, and cache write), and the energy consumption of the write intensity predictor (only for Dynamic PHC).

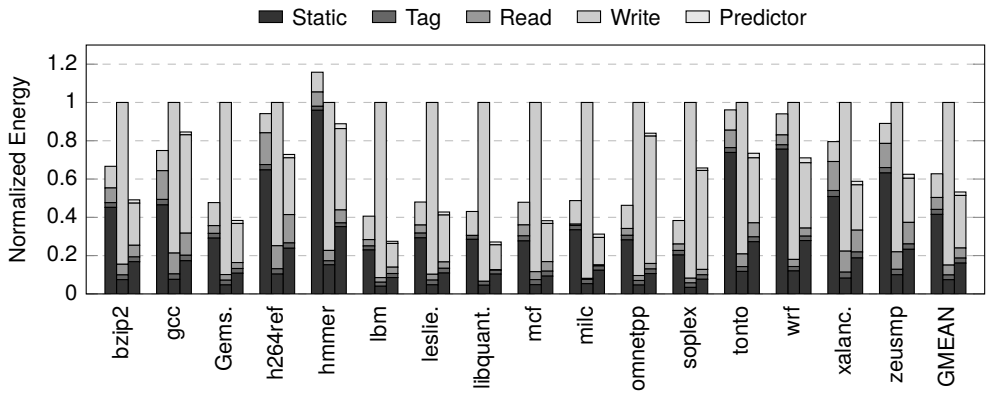


Figure 5.9: Energy breakdown of the SRAM/STT-RAM baseline (left/middle) and Dynamic PHC (right).

In the SRAM baseline, the high static power of SRAM contributes the most to L2 cache energy. Although replacing SRAM with STT-RAM mitigates this issue, it also significantly increases write energy consumption. Dynamic PHC balances these two aspects by (1) constructing caches with large STT-RAM and small SRAM to minimize static power and (2) judiciously placing cache blocks to SRAM in a way to maximize the benefit of low write energy of SRAM. On average, Dynamic PHC consumes 61% less static energy compared to the SRAM baseline and 61% less dynamic energy compared to the STT-RAM baseline.

5.5.3 Coverage and Accuracy

Figure 5.10 shows coverage and accuracy of the write intensity predictor with and without dynamic threshold adjustment. Legends that start with 'True/False' indicate correct/incorrect prediction. Also, legends that end with 'Positive/Negative' represent cases where blocks are predicted to be write-intensive/non-write-intensive. For example, 'True Positive' represents the case where a block is predicted to be write-intensive at block fill, and its cost at eviction is actually greater than or equal to the threshold. On average, the write intensity predictor achieves 93% of accuracy under the static threshold policy and using dynamic threshold adjustment maintains about the same level of accuracy.

One interesting observation is that the optimal ratio of blocks allocated to the SRAM region (83%) is much higher than the ratio of the SRAM region capacity (25%). One might expect the two ratios to be about the same in order to balance the capacity pressure

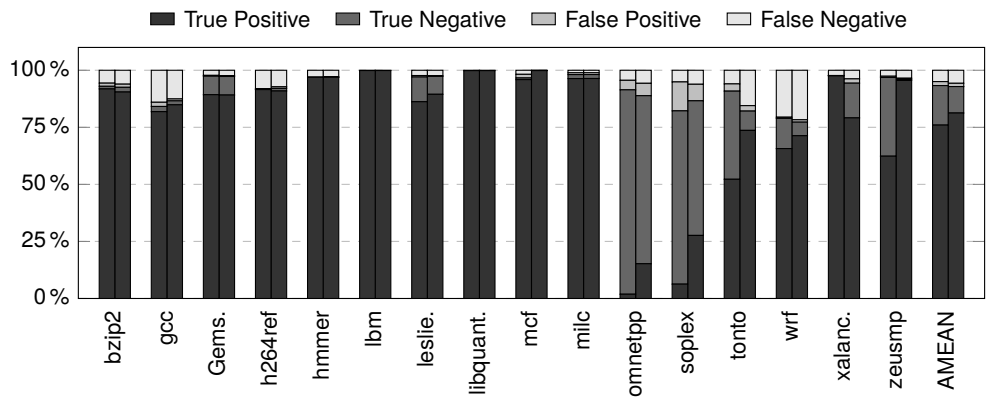


Figure 5.10: Coverage and accuracy of the write intensity predictor with static (left) and dynamic (right) thresholds.

of two regions, i.e., maintaining the ratio of blocks inserted into the SRAM region to that of the STT-RAM region to be the capacity ratio of the two. However, we found that allowing more blocks to be allocated into the SRAM region does not increase miss rates until a certain point because most of the blocks are reused within a short time interval, in which case early eviction of them does not increase miss rates. This is the reason why Dynamic PHC is based on monitoring miss rates, rather than simply controlling the insertion ratio of two regions.

5.5.4 Sensitivity to Write Intensity Threshold

Figure 5.11 shows total energy consumption (the L2 cache and the main memory) in Static PHC with different thresholds from −20 to 50. The results are normalized to those of Dynamic PHC, which are shown as a thick line.

As can be seen in the figure, the best value for the write intensity threshold widely varies across applications. For example, the best threshold for bzip2 is −5, while that for soplex is 40. This signifies the need for dynamic threshold adjustment, especially considering that static profiling is not always possible and static thresholds are less robust against runtime variation.

5.5.5 Impact of Dynamic Set Sampling

Our architecture uses the concept of dynamic set sampling for both sampler and threshold selector to reduce storage overhead. To evaluate its impact on prediction accuracy, we compare total energy consumption (excluding the write intensity predictor) without

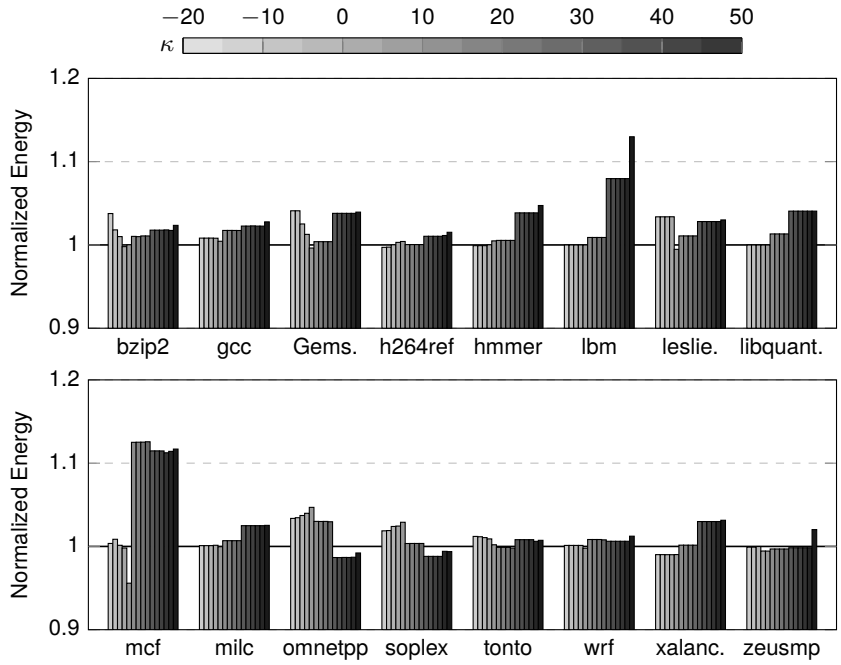


Figure 5.11: Total energy consumption of Static PHC with different thresholds (normalized to Dynamic PHC).

dynamic set sampling against that with it. According to the evaluation, the two differs by only 1.3% on average while dynamic set sampling reduces storage overhead by $1/32$.

5.5.6 Results for Non-Write-Intensive Workloads

Figure 5.12 shows energy consumption of the L2 cache and the main memory for 13 non-write-intensive benchmarks in SPEC CPU2006. The results are normalized to the STT-RAM baseline.

As opposed to Figure 5.8, the SRAM baseline consumes 73% higher energy in the L2 cache compared to the STT-RAM baseline. This is because, due to the low write intensity of the benchmarks, static energy becomes the dominant source of the energy consumption of the L2 cache.

Under this circumstance, our architecture achieves almost the same energy consumption compared to the STT-RAM baseline. This, together with the results in Figure 5.8, can be summarized as follows:

Our architecture reaches the energy consumption of an SRAM cache under high write intensity and that of an STT-RAM cache under low write intensity.

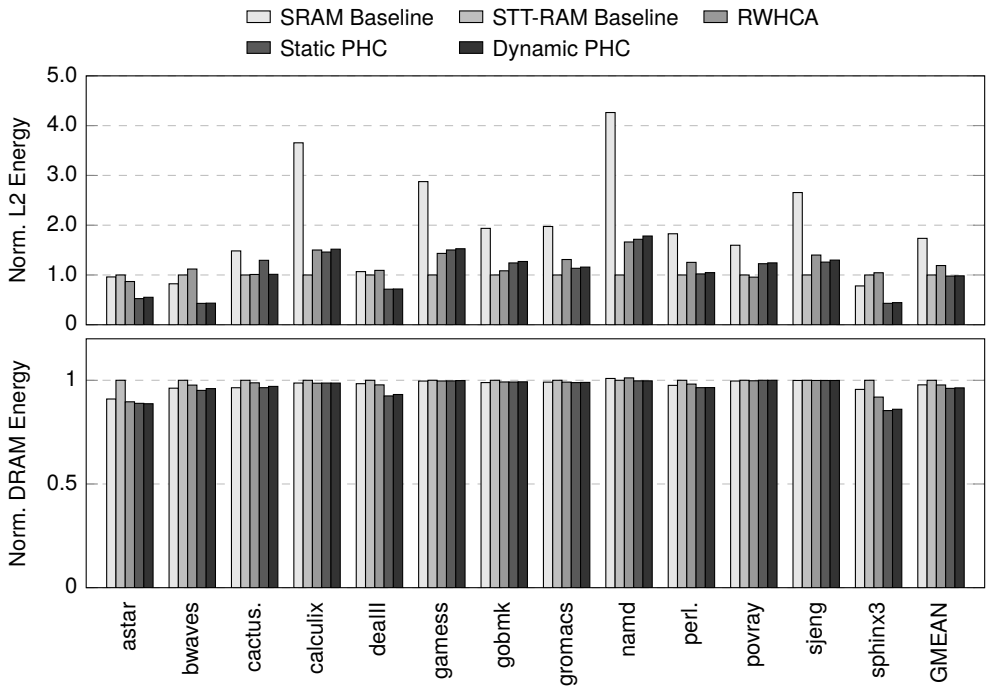


Figure 5.12: Energy consumption of the L2 cache and the main memory for benchmarks with low write intensity.

This is an extremely desirable property since it takes advantage of both technologies with runtime adaptation. On average across all SPEC CPU2006 workloads, our scheme achieves 29% and 30% reductions in energy consumption of the L2 cache compared to the SRAM baseline and the STT-RAM baseline, respectively, while those of RWHCA are only 2.3% and 3.2%.

5.6 Multicore Evaluations

In this section, we evaluate our architecture under the quad-core system described in Section 5.4.1. Simulation configurations are similar to those in the single-core evaluations, except the four times larger capacity of the L2 cache to keep the same per-core L2 cache capacity. To reflect this larger L2 cache configuration, ΔE_{r} and ΔE_{w} are different from the ones in the single-core system. Due to this, we choose the best write intensity threshold for Static PHC from $\kappa = -12$ to 32 at intervals of 4, instead of using the same range of possible thresholds as in the single-core evaluations.

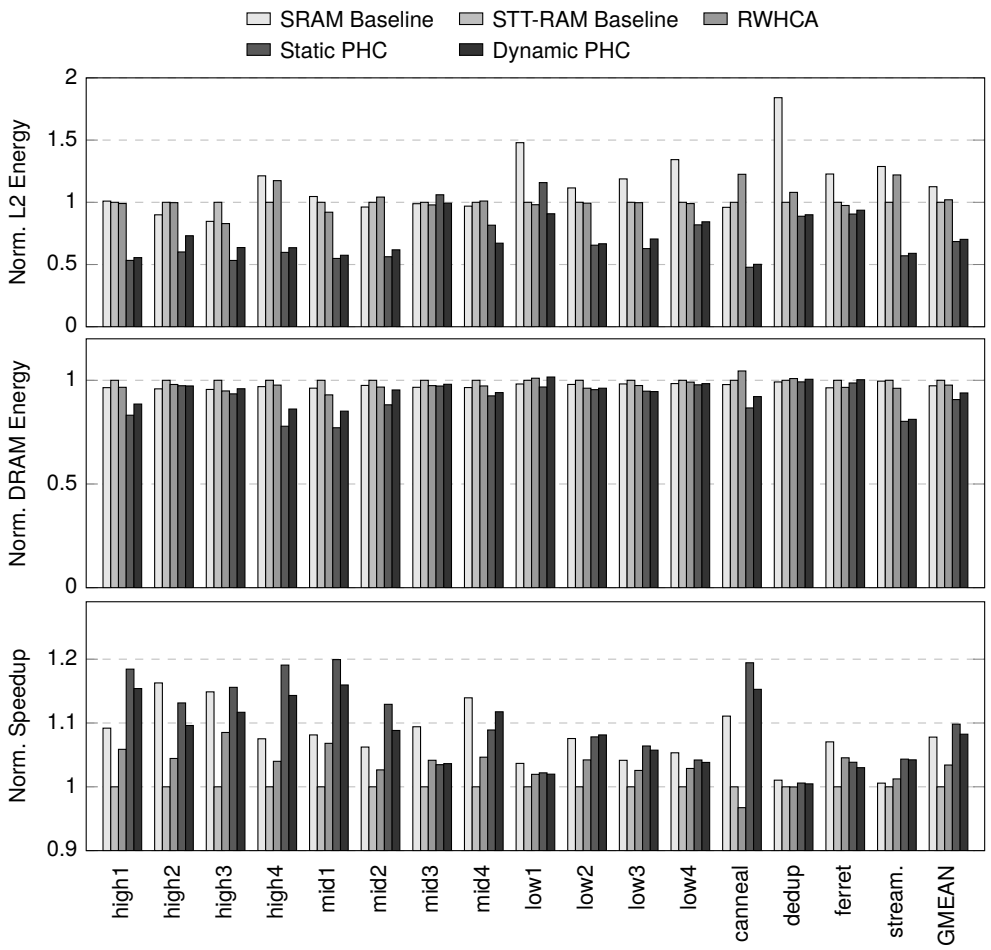


Figure 5.13: Energy consumption and speedup under the quad-core system (normalized to the STT-RAM baseline).

Figure 5.13 compares the energy consumption and speedup of our architecture against previous approaches. The results are normalized to the STT-RAM baseline. We use *weighted speedup* [105] as a performance metric for multiprogrammed workloads.

As opposed to the single-core evaluations shown in Figure 5.8, the SRAM baseline now consumes more energy than the STT-RAM baseline does. This is because the quadrupled capacity of the L2 cache increases the portion of static energy in its energy consumption. Similarly, the effectiveness of RWHCA in energy reduction degrades as well due to the increased static power from the SRAM region.

Nevertheless, Dynamic PHC still achieves a 30% reduction in the energy consumption of the L2 cache compared to the STT-RAM baseline. This, in turn, leads to a 38% smaller energy consumption in the L2 cache compared to the SRAM baseline. These results are achieved through fully utilizing a small SRAM cache (which minimizes increase in static energy compared against the SRAM baseline) to reduce write energy consumption in the STT-RAM region. In addition, the energy consumption of the main memory in Dynamic PHC is 6.1% and 3.6% smaller than that of the STT-RAM baseline and the SRAM baseline, respectively.

In addition, Dynamic PHC improves system performance by 8.2% compared to the STT-RAM baseline. This is contributed mainly by the reduction in write operations to the STT-RAM region as in the single-core evaluations. However, the amount of speedup is higher in the quad-core system since the L2 cache suffers from higher contention in the quad-core system than in the single-core system, which can be alleviated in our architecture by reducing write operations to the STT-RAM region.

As in the single-core evaluations, Static PHC and Dynamic PHC show negligible difference in terms of energy and performance under the quad-core system. This is an important result in the context of multi-core systems where static profiling of workloads is impractical due to runtime variation.[11]

Prediction accuracy in the quad-core system remains almost the same as that in the single-core system. On average, Static PHC and Dynamic PHC achieve 96% and 95% of accuracy, respectively.

5.7 Summary

We proposed prediction hybrid caches, a STT-RAM based hybrid cache architecture with a novel block placement policy. The key concept is write intensity prediction, which predicts write intensity of cache blocks on their misses and determines block placement based on it. For this purpose, our mechanism utilizes a newly discovered correlation between write intensity of blocks and instruction addresses that incur their misses. Moreover, it includes a mechanism to dynamically adjust the threshold of write intensity according to application characteristics. Experimental results show that our architecture achieves 28% (31%) energy reductions in hybrid caches, 3% (4%) energy reductions in main memory, and 1.4% (4.7%) speedup in the single-core (quad-core) system compared to the existing hybrid cache architecture. Although this work utilizes

[11] In the case of multiprogrammed workloads, combinations of applications to be run simultaneously are hard to be known in advance. Similarly, actual scheduling of threads in multithreaded workloads highly depends on runtime information.

the concept of write intensity prediction for hybrid caches, we believe that it can also be adopted for many other non-volatile memory systems where prior knowledge of write-intensive data could be helpful for improving energy efficiency.

Chapter 6

Dead Write Prediction Assisted STT-RAM Cache

As introduced in the previous chapter, one common approach to reducing the write energy consumption of STT-RAM caches is to utilize traditional SRAM caches (or buffers) along with STT-RAM caches and let them serve some of write operations [31–33, 35, 36]. Although we demonstrated that our approach is very effective in reducing the energy consumption of STT-RAM caches, its effectiveness is mainly constrained by the small SRAM size, which cannot be arbitrarily increased due to the high static power of SRAM. Even if the SRAM size is carefully determined, it may consume higher energy than a pure STT-RAM cache does under infrequent write requests because of the increased static power from SRAM.

In this chapter, we propose a new architectural technique to reduce the write energy consumption of STT-RAM last-level caches, which does not require extra SRAM caches. We exploit our observation that a substantial amount of write operations to last-level caches (incurred by writebacks from lower-level caches or block fills), which we call *dead writes*, can actually bypass the caches without any extra cache misses. This provides a new opportunity to avoid unnecessary write energy consumption. To this end, we propose *Dead Write Prediction Assisted STT-RAM Cache Architecture (DASCA)*. The key idea of DASCA is to identify dead writes and bypass them to reduce the write

This chapter is originally published in Proceedings of the International Symposium on High Performance Computer Architecture [106].

energy consumption. Since identifying dead writes requires future knowledge of cache access patterns, we design a dead write predictor that predicts whether a write operation would be a dead write or not.

This chapter makes the following contributions:

- We propose the concept of dead writes and their classification (composed of three types) as a theoretical model for redundant write elimination.
- We rethink the use of dead block prediction in the context of STT-RAM caches. While traditional dead block prediction has aimed at reducing miss rates, we utilize it to reduce the write energy consumption of STT-RAM caches.
- We propose an STT-RAM last-level cache design that predicts and bypasses dead writes for write energy reduction. Whether to bypass a write operation or not is determined by our dead write predictor based on the state-of-the-art dead block predictor [88].
- We evaluate the effectiveness of our architecture and show that it significantly improves the performance and energy consumption of STT-RAM last-level caches. Our architecture also reduces off-chip main memory accesses, thereby improving the energy efficiency of main memory as well.

6.1 Motivation

6.1.1 Energy Impact of Inefficient Write Operations

Figure 6.1 shows the energy consumption of SRAM and STT-RAM caches when they are used as an L2 cache of a two-level cache hierarchy (refer to Section 6.4.1 for the detailed configuration). The results are broken down by static energy, dynamic energy of the tag array, and read and write energy of the data array. The rightmost bars labeled by 'GMEAN' indicate the geometric mean of individual components over 16 workloads.

We observed the energy consumption of the SRAM cache is dominated by its high static power and the STT-RAM cache is very effective to alleviate such waste. However, high write energy of the STT-RAM cache actually offsets this advantage in most cases thereby increasing the total energy consumption.

Most of the existing solutions to this problem have focused on identifying cache blocks that are written many times (many writebacks) until their eviction (also known as write-intensive blocks) and reducing write energy for such blocks [31–33, 35, 36, 44]. However, there are actually two kinds of events that incur write operations to the

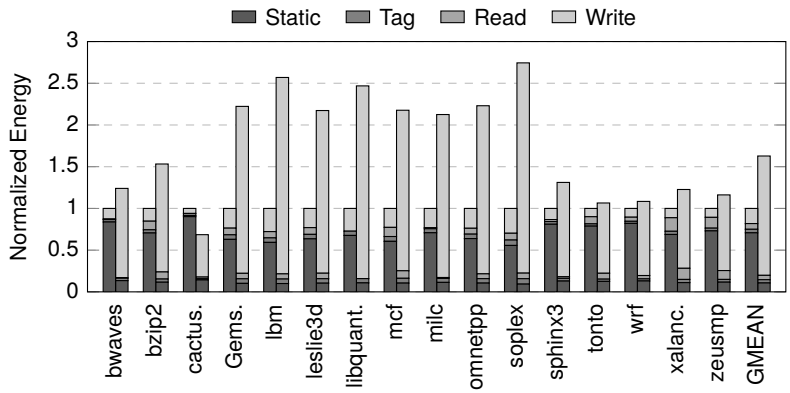


Figure 6.1: Energy consumption of SRAM (left) and STT-RAM (right) L2 caches.

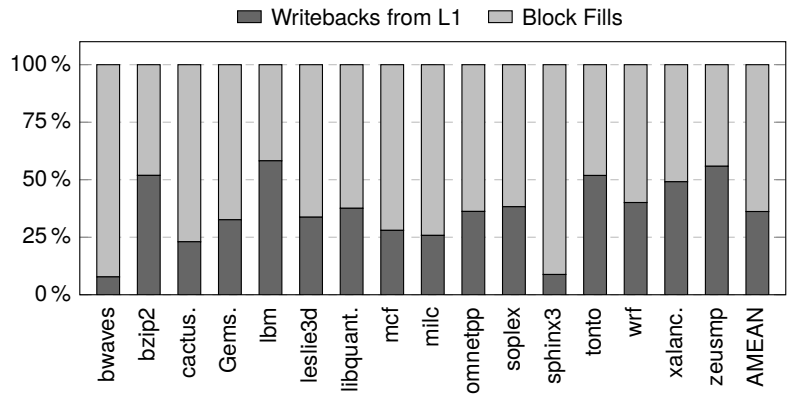


Figure 6.2: Breakdown of write-inducing events in the L2 cache.

data array of an L2 cache: (1) writeback from the L1 cache and (2) block fill into the L2 cache on a read miss. Between the two, our evaluation shows that the latter actually contributes more to write energy consumption than the former does as shown in Figure 6.2. Furthermore, deeper cache hierarchies make the portion of block fills even larger in last-level caches since repeated writebacks to the same cache block are filtered by multiple levels of caches. This advocates the need for a new technique that can effectively reduce both events from STT-RAM last-level caches.

6.1.2 Limitations of Existing Approaches

Hybrid caches [31–33, 35, 36] are the most well-known approaches to reducing the write energy consumption of STT-RAM caches. However, one trivial shortcoming of such

proposals is the increased static power consumption due to SRAM caches, which may easily offset the benefit of their low dynamic energy especially when the cache capacity is large. Moreover, they are not that effective in reducing write energy consumed by block fills due to the limited capacity of SRAM caches. Our evaluation will show that using hybrid caches for large last-level caches is not as energy-efficient as our technique due to these inherent limitations.

Instead of using SRAM caches, several studies have tried to design STT-RAM cells for lower write energy at a cost of shortened retention time down to several milliseconds [43–45]. Such *volatile* STT-RAM cells now require periodic scrubbing to keep data alive as in DRAM cells, thereby imposing a new challenge in STT-RAM cache design. Although this approach is effective in reducing write energy of STT-RAM caches, it may not be suitable for STT-RAM last-level caches due to the following two reasons. First, scrubbing overhead increases as the cache capacity gets larger, which limits its scalability. Second, such overhead is exacerbated due to the long lifetime of last-level cache blocks.

6.1.3 Potential of Dead Writes

Our objective is to design an STT-RAM last-level cache architecture that can minimize write activities incurred by both writebacks from the L1 cache and block fills. To achieve this goal, we take advantage of the following observation: *A substantial amount of write operations (either block fills or writebacks from lower-level caches) can bypass the last-level caches without incurring extra cache misses.* We call such write operations *dead writes*. A simple example of dead writes is fill operations for blocks that will never be accessed again. Since bypassing avoids writing data to caches, accurately predicting and bypassing dead writes can reduce the write energy consumption of STT-RAM caches without sacrificing miss rates.

Figure 6.3 shows the potential of eliminating dead writes. Perfect elimination of all dead writes based on oracle prediction could reduce 89% of the write energy consumption on average, thereby bridging the huge gap of write energy between SRAM and STT-RAM. It could reduce write energy consumed by block fills (96% reduction) as well as writebacks from lower-level caches (81% reduction). However, it is impossible to completely eliminate dead writes because the *future* access pattern of a cache block is required to determine whether bypassing the block would increase the miss rate or not. In the next sections, we describe all possible cases of dead writes and present a practical solution that can accurately predict such dead information.

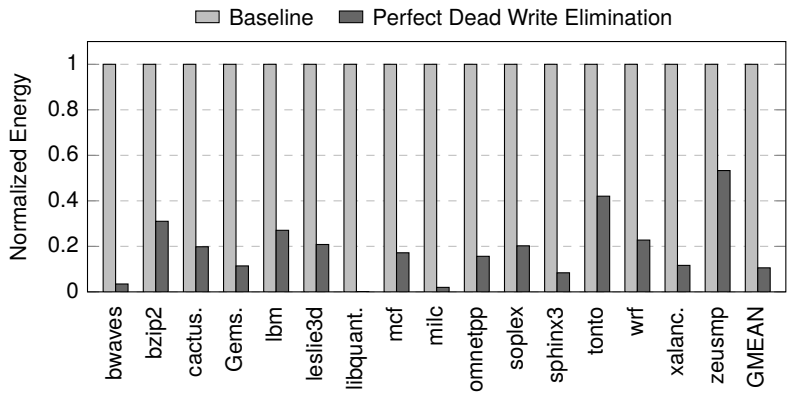


Figure 6.3: Normalized write energy after perfect elimination of dead writes (impractical).

6.2 Dead Write Classification

In this section, we classify all possible scenarios for dead writes into three cases: dead-on-arrival fills, dead-value fills, and closing writes. Figure 6.4 shows an example sequence of accesses to the same cache set for each case. In this example, we use R_A and W_A to represent a read operation and a write operation to block A, respectively, and assume a four-way set-associative cache with the LRU replacement policy.

6.2.1 Dead-on-Arrival Fills

It is well known that a considerable amount of cache blocks are dead on arrival, that is, they are accessed only once on their read misses during their lifetime [87, 88, 107, 108]. One important characteristic of such blocks is that bypassing fill operations for them does not increase the miss rate at all as the blocks will not be accessed again until eviction. This indicates that such fill operations, which we call *dead-on-arrival fills*, belong to dead writes. A fill operation to a cache block can be dead-on-arrival in one of the following two cases: (1) The corresponding memory region is never accessed again after the fill during the program execution, or (2) too many blocks in the same cache set are accessed between two accesses to the block. The two cases are also known as a streaming access pattern and a thrashing access pattern, respectively [108].

Figure 6.4a shows an example of dead-on-arrival fills (thrashing access pattern). In this example, block A is accessed only once during its first residence period between the first R_A and R_E. Thus, the fill operation of block A at the first R_A is considered as a dead-on-arrival fill.

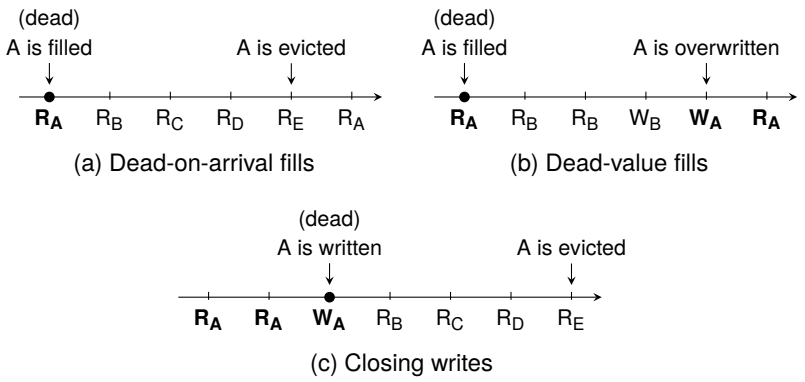


Figure 6.4: Three different classes of dead writes.

6.2.2 Dead-Value Fills

Consider a case where a block receives a writeback request from lower-level caches right after the block is filled by a read miss, but before any read operation, i.e., the filled block data are overwritten before being read. In this case, the data of the block do not need to be filled on its miss as they will soon be overwritten by the subsequent writeback. This is the second case of dead writes, which is called *dead-value fills*. This happens when a lower-level cache block becomes dirty before its first eviction during the lifetime of the corresponding last-level cache block.

Note that dead-value fills should be on a read miss, not on a write miss, in order to be dead writes. This is because, if we bypass a block fill on a write miss, which is followed by a write operation, there would be two writebacks (one bypass and one from the last-level cache) to the main memory in total, whereas, without bypass, only one writeback from the last-level cache is needed.

Figure 6.4b illustrates an example of dead-value fills. In this example, the fill operation at the first $\mathsf{R_A}$ is a dead-value fill since there are no read operations to block A between the first $\mathsf{R_A}$ and the subsequent $\mathsf{W_A}$.

6.2.3 Closing Writes

Let us assume that an access sequence of a block during its lifetime is ended with a write operation incurred by a writeback request. Then, it is safe to bypass the last write operation as it will be written back anyway on its eviction. This is the third case of dead writes, which is called *closing writes*. This type of dead writes occurs very frequently in

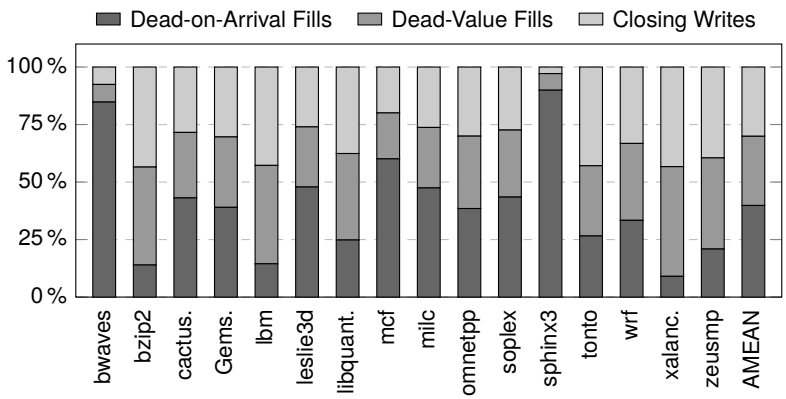


Figure 6.5: Breakdown of three dead write classes.

writeback cache hierarchies as the writeback policy delays write requests until block eviction.[1]

Note that a write operation could never be a closing write if any operation comes after it during its lifetime. For example, let us assume that a block receives a read, a write, and a read operation in sequence until its eviction. In this case, although the write operation is the last write operation for this block, it is not a closing write as there is a subsequent read operation.

Figure 6.4c shows an example sequence of accesses for closing writes. Block A receives three operations during its lifetime and the last one is a write operation. Therefore, this write operation belongs to closing writes.

6.2.4 Decomposition

Figure 6.5 shows the breakdown of three dead write classes in the 1 MB L2 cache of our baseline configuration (see Section 6.4.1). On average, all three classes are almost evenly distributed. This indicates that bypassing dead writes would effectively reduce write energy consumed by block fills as well as writebacks from lower-level caches (the former has not been addressed well in previous work).

In some cases, the distribution is biased toward one specific class. In particular, most of the dead writes are dead-on-arrival fills in bwaves and sphinx3. This is because these two workloads have relatively low hit rates and a small number of writebacks. On

[1] In some cases, write-through caches are used to simplify cache coherence protocols and this claim may not hold for such cache hierarchies. However, even in such cases, caches right above the last-level caches usually adopt the writeback policy to hide a large amount of write traffic from the large, energy-consuming last-level caches.

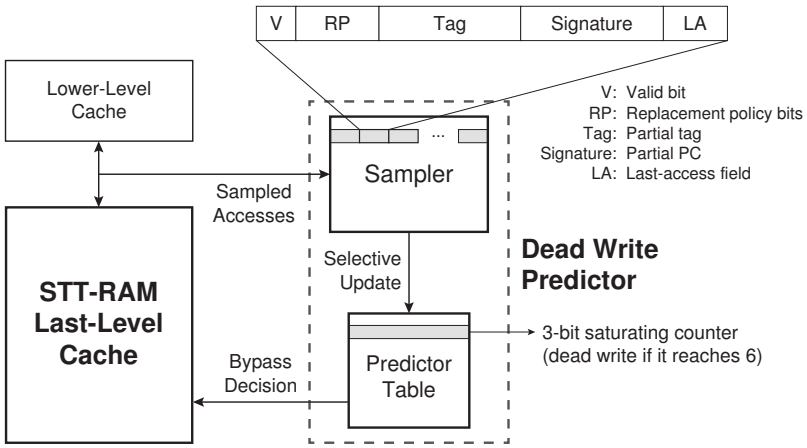


Figure 6.6: Overview of DASCA.

the other hand, dead-on-arrival fills take only a small portion of dead writes in bzip2, lbm, and xalancbmk as those workloads have higher hit rates, indicating many blocks are re-referenced frequently.

6.3 Dead Write Prediction Assisted STT-RAM Cache Architecture

Figure 6.6 shows an overview of DASCA, the proposed cache architecture for write energy reduction. It has a dead write predictor that detects dead writes without future knowledge. Predicted dead writes bypass the cache in both directions—writebacks from lower-level caches and block fills—as discussed earlier. Our architecture assumes non-inclusive caches by default, but we also propose a bypassing scheme that supports both inclusive and non-inclusive caches. This section describes how to predict and bypass dead writes in detail.

6.3.1 Dead Write Prediction

The proposed dead write predictor is based on existing dead block predictors. This is because dead write prediction shares some similarities with dead block prediction in the sense that both predict future access patterns of cache blocks. However, there are also important differences that need to be carefully addressed. In this section, we describe the baseline of our predictor design and our choices to re-architect the baseline for accurate prediction of dead writes.

Baseline. As our baseline, we choose the sampling dead block predictor [88], a state-of-the-art dead block predictor with modest storage overhead. The sampling dead block predictor is a PC-based predictor that correlates dead blocks with instruction addresses (called *signatures*) that cause the cache accesses. The intuition behind this is that, if a block is dead after a cache access with a given signature, future accesses having the same signature will also make other blocks dead. To reduce storage and energy overhead, it samples a few cache sets ($\sim 1/64$ of the entire sets) from the cache and keeps track of PC information only for those sets. This is accomplished by keeping a separate partial tag array called sampler as shown in Figure 6.6. For each cache access to sampled sets, it simulates block replacements for those sets based on the replacement policy of the cache.

Predictions are made by the predictor table, which consists of saturating counters similar to branch predictors. The counters are indexed by the signatures stored in sampler entries. The update policy of the predictor table will be explained later in this section.

Signature. To predict dead-on-arrival fills and dead-value fills during fill operations, we use the same signature as the one used in the baseline. For the implementation of the predictor, the PC information is sent along with the miss load request on each lower-level cache miss.

In contrast, to predict closing writes during writeback operations, we propose to use a different signature, which is the last-touch instruction address of the block to be written back. The reason why a new signature is needed is that the baseline signature shows less correlation with writeback requests. Consider a case where instruction X incurs a miss on a lower-level cache, and thus, makes dirty victim A evicted. In this case, the baseline uses X's address as a signature for A's writeback. On the other hand, we use the address of instruction Y that accessed block A for the last time as the signature. Between the two schemes, evaluation shows that the latter outperforms the baseline because the behavior of block A relies more on the instructions that have accessed the block than the instruction that incurs eviction of the block.

To keep track of the last-touch instruction addresses, each lower-level cache tag is equipped with an additional partial PC field. This field is updated on each access only if the block is dirty because the new signature is used only for writebacks. Stored partial PCs are transferred as part of writeback requests.

How to Update the Predictor Table? To identify dead writes, we add a two-bit last-access field for each sampler entry to store the type of the last access. It can be one of four types: read hit, read miss, write hit, and write miss. This information enables the

predictor to determine whether the last access to the block was a dead write or not. In particular, the three types of dead writes can be detected exactly as follows from their definitions:

- When a sampler entry is evicted, the last access to the entry is a dead-on-arrival fill if the state of the last-access field is read miss.
- When a write hit occurs on an existing sampler entry, the last access to the entry is a dead-value fill if the state of the last-access field is read miss.
- When a sampler entry is evicted, the last access to the entry is a closing write if the state of the last-access field is either write hit or write miss.

Therefore, we increment (toward dead) the counter in the predictor table indexed by the signature of the last access if one of the above conditions is satisfied. Otherwise, we decrement (toward live) the counter by default.

When to Update the Predictor Table? The baseline updates the predictor table on every cache access or eviction in the sampled sets (*full update policy*). This makes sense for dead block prediction because cache blocks could become dead after any kind of cache accesses. However, the situation is different for dead write prediction since not all types of cache accesses are eligible to be dead writes. In particular, read hits can never be dead writes and prediction is not needed for such operations.

Consequently, we propose the *selective update policy*, which does not update the predictor table when the last access was a read hit. Note that the last-access field provides this information. An important implication of using this policy is that the predictor now implicitly utilizes the temporal information of cache accesses in addition to signatures. For example, let us assume that there is a memory read instruction, which is always a dead write if it is the first access to the block, but never becomes a dead write otherwise. In such cases, dead writes cannot be identified accurately with the full update policy as both cases update the same predictor entry thereby interfering with each other. On the contrary, the selective update policy can predict the former case as a dead write since the predictor is neither updated nor accessed in the latter case.

In addition to that, the selective update policy mitigates pressure on the predictor table due to fewer updates. This reduces aliasing in the predictor table, which slightly improves the prediction accuracy.

Putting All Together. Table 6.1 summarizes our update mechanism. It shows the direction of counter updates on a specific event depending on the type of the last access

Table 6.1: Predictor Update Mechanism

Last Access	Event	Update	Reason
Read Miss	Write Hit	Dead	Dead-value fill
Read Miss	Read Hit	Live	Not a dead fill
Write	Hit	Live	Not a closing write
Read Hit	Hit	Ignore	Read hit
Read Miss	Eviction	Dead	Dead-on-arrival fill
Write	Eviction	Dead	Closing write
Read Hit	Eviction	Ignore	Read hit

to the entry. For example, the first line represents that, when a write hit occurs on a sampler entry, we update the counter toward dead if the last access to the entry was a read miss.

6.3.2 Bidirectional Bypass

The next step is to bypass predicted dead writes for write energy reduction. We call our scheme *bidirectional bypass* since bypass happens in two directions, from main memory to lower-level caches (upward bypass) and vice versa (downward bypass). Note that our scheme bypasses the last-level cache only. It does not bypass the sampler but updates it as necessary in order to check later to see if bypassed accesses are actually dead writes and update predictor entries on a misprediction.

The upward bypass is to skip block fills that are dead writes, that is, dead-on-arrival fills and dead-value fills. It simply hands over the data loaded from main memory to lower-level caches without inserting a cache block into the last-level cache. Note that it does not increase the miss rate at all as long as the dead write prediction is perfect. For dead-on-arrival fills, the blocks are no longer accessed, and thus, bypassing them does not affect the miss rate. Also, for dead-value fills, the blocks will be inserted into the cache by the subsequent write operations *before any other cache hits*. Thus bypassing such blocks also does not affect the miss rate, since the subsequent write misses do not need main memory accesses under the write-allocate policy when the block sizes are the same.[2]

The downward bypass is to skip dead writeback requests, or closing writes. For this, the last-level cache should be checked beforehand to see if the target cache block resides

[2]We assume the same block size for the last-level cache and caches right above it, which is a very common choice.

in the cache and invalidate it if does. This is necessary to prevent from reading stale data. After that, the writeback requests can be forwarded directly to main memory as in write-through caches. Note that this does not incur any extra writebacks from the last-level cache under perfect prediction as closing writes indicate no more accesses to the cache blocks by definition.

Our bypass scheme can be easily integrated into non-inclusive cache hierarchies. On the other hand, when a system enforces the inclusion property between the last-level cache and the lower-level caches, it is impossible to simply adopt our scheme because the upward bypass inevitably violates the inclusion property. One simple solution is to insert the blocks into the LRU slot instead of bypassing them [93]. Although this shows a similar effect in miss rate reduction, it is rather helpless for write energy reduction because it inserts the cache blocks anyway. Thus, we propose to introduce an additional cache block state called *void* for inclusive cache hierarchies. Instead of the upward bypass, we insert the block into the LRU slot *without writing its data* and mark it as void. Accesses to void blocks are handled just as in cache misses to fill the missing data. However, coherence state bits of void blocks are updated as if the blocks are valid. This simple modification facilitates write energy reduction through bypass while maintaining the inclusion property.

6.4 Evaluation Methodology

6.4.1 Simulation Configuration

We evaluate our architecture using a cycle-accurate x86-64 simulator based on Pin [95]. The simulator is capable of modeling asymmetric read/write latencies and bank contention of STT-RAM caches so as to take their long write latency into account. The baseline configuration is a single- or quad-core system operating at 3 GHz. We model out-of-order, four-issue superscalar processors with 128-entry instruction window.

The cache hierarchy of the system comprises 32 KB, 4/8-way set-associative L1 instruction/data caches and a 1 MB per core, 16-way set-associative, shared L2 cache, all having 64-byte blocks and the LRU replacement policy. The L1 data cache and the L2 cache are non-blocking caches supporting up to eight in-flight misses. For the quad-core system, we maintain cache coherence with the snoop-based MESI protocol without enforcing the inclusion property.

Table 6.2 shows the energy and delay characteristics of SRAM and STT-RAM caches under 45 nm technology modeled by CACTI 6.5 [13] and NVSim [14]. STT-RAM cell parameters are extracted from the previous studies [15, 99] and fed into NVSim. Both

Table 6.2: Characteristics of SRAM and STT-RAM Caches

	Single-Core (1 MB)		Quad-Core (4 MB)	
	SRAM	STT-RAM	SRAM	STT-RAM
Read Latency (ns)	3.18	2.83	4.32	3.81
Write Latency (ns)	3.18	12.01	4.32	13.09
Read Energy (nJ)	0.08	0.07	0.14	0.08
Write Energy (nJ)	0.08	0.64	0.14	0.69
Static Power (mW)	14.63	2.32	62.65	7.80

SRAM and STT-RAM caches use SRAM tag arrays since status bits and replacement policy bits of tags could be updated frequently [85, 94] and static power of the SRAM tag array accounts for only 3% of the energy consumption of the STT-RAM L2 cache under our configuration. ITRS LOP cells are used for peripherals and SRAM cells since, according to the previous study [98], the characteristics of last-level caches for modern high-performance cores are known to best match LOP devices among three device models supported by CACTI.

For the dead write predictor, every 32nd L2 cache set is sampled to construct the 16-way set-associative sampler, i.e., 32/128 sets in total for the 1/4 MB caches in the single-/quad-core system. Each sampler entry has a valid bit, a 4-bit LRU information, a 16-bit partial tag, a 16-bit signature, and a 2-bit last-access field. The predictor table is composed of 8192 3-bit saturating counters and their threshold is set to six. In addition, each L1 cache tag is equipped with a 16-bit partial PC field to store the last-touch instruction addresses as discussed in Section 6.3.1. In total, the storage overhead of a dead write predictor is only 6.44/13.75 KB for the 1/4 MB cache, which is negligible compared to the size of the last-level cache. Such overhead is also modeled by CACTI 6.5 [13]. We verified that the sampler and the predictor table are not on the critical path and thus do not increase the access latency of the last-level cache.

Our system has 4 GB main memory configured as having two channels, one rank per channel, eight banks per rank, and 8 KB row buffers with the timing parameters of DDR3-1600 11-11-11-28 [96]. The memory controller uses the FR-FCFS scheduling under the open-row policy [97] and has a 64-entry request queue. The power consumption of main memory is modeled by Micron System Power Calculator [96, 100].

Table 6.3: Workloads from SPEC CPU2006

Workload	WPKI
mcf	96.1
lbm	75.2
soplex	54.4
libquantum	50.2
GemsFDTD	32.5
leslie3d	32.0
omnetpp	29.7
milc	29.4
sphinx3	13.3
zeusmp	13.0
bwaves	12.4
xalancbmk	11.4
wrf	10.9
bzip2	10.1
tonto	8.8
cactusADM	7.3

Mix	Workloads
high1	mcf, lbm, soplex, libquantum
high2	GemsFDTD, leslie3d, omnetpp, milc
high3	mcf, soplex, GemsFDTD, omnetpp
high4	lbm, libquantum, leslie3d, milc
mid1	mcf, lbm, cactusADM, tonto
mid2	soplex, libquantum, bzip2, wrf
mid3	GemsFDTD, leslie3d, xalancbmk, bwaves
mid4	omnetpp, milc, zeusmp, sphinx3
low1	cactusADM, tonto, bzip2, wrf
low2	xalancbmk, bwaves, zeusmp, sphinx3
low3	cactusADM, bzip2, xalancbmk, zeusmp
low4	tonto, wrf, bwaves, sphinx3

6.4.2 Workloads

For the single-core system, we select 16 write-intensive benchmarks from SPEC CPU2006 [101] as shown in Table 6.3, where write-intensiveness is defined as write per kilo instruction (WPKI)[3] of the last-level cache. For benchmarks having multiple input sets, we select the representative ones by referring to the previous work [102]. We use PinPoints [103] to select and run a representative phase of each benchmark for one billion instructions. We also evaluated the other 13 benchmarks from the SPEC suite and found that the vanilla STT-RAM cache already consumes 41% lower energy than the SRAM cache does and our architecture consumes even lower energy (23% lower on average compared to the vanilla STT-RAM cache).

For the quad-core system, we construct 12 multiprogrammed workloads from the single-core workloads. To evaluate a wide variety of benchmarks, we first categorize the 16 single-core benchmarks into top eight write-intensive benchmarks (denoted as HIGH) and the other eight benchmarks (denoted as Low). Then we form four high intensity

[3]In this context, the term 'write' covers read misses (block fills) as well as write hits/misses (writebacks from the L1 cache).

mixes with four HIGHS each, four medium intensity mixes with two HIGHS and two Lows each, and four low intensity mixes with four Lows each as shown in Table 6.3. We run the multiprogrammed workloads for four billion instructions in total.

In addition to that, we use four multithreaded benchmarks from the PARSEC benchmark suite [104]. Each benchmark is run using the simlarge input set for a maximum of two billion instructions from the beginning of Regions-of-Interest (ROI).

6.5 Evaluation for Single-Core Systems

For the single-core system, we compare the effectiveness of our architecture with Region-based Hybrid Cache Architecture (RHCA) [32] and an STT-RAM cache with Sampling Dead Block Predictor [88]. RHCA is configured to have twelve STT-RAM ways and four SRAM ways per set. Since RHCA assumes that both read and write operations of STT-RAM caches are slower than those of SRAM caches, which is not true under our configuration, we add a modified version of RHCA in which reading STT-RAM blocks does not trigger migrations to the SRAM region.

6.5.1 Energy Consumption and Speedup

Figure 6.7 compares energy consumption and speedup of our architecture against five different configurations. The results are normalized to those of the STT-RAM baseline. The energy consumption of the L2 cache includes energy overhead of dead write prediction, if any, including the dead write predictor and the partial PC fields added to L1 cache tags.

Compared to the STT-RAM baseline, DASCA reduces the energy consumption of the L2 cache by 68% on average with low energy overhead from the dead write predictor (4.2% of the energy consumption of the L2 cache). More importantly, it consumes 47% lower energy compared to the SRAM baseline, whereas the STT-RAM baseline actually consumes 63% *higher energy* than the SRAM baseline does due to the eight times higher write energy of the STT-RAM cache as shown in Table 6.2. This clearly shows the importance of techniques for reducing write energy of STT-RAM caches, which is accomplished well through our scheme.[4]

An interesting observation is that DASCA concentrates more on reducing write energy consumed by block fills. On average, it reduces 93% of write energy consumed

[4]We also evaluated DASCA for SRAM caches and found that it reduces energy consumption by 15% with 4.2% performance improvement compared to the SRAM baseline. The reason of the lower energy reduction compared to DASCA for STT-RAM caches is that write energy does not account for a major portion in SRAM caches as discussed in Section 6.1.1.

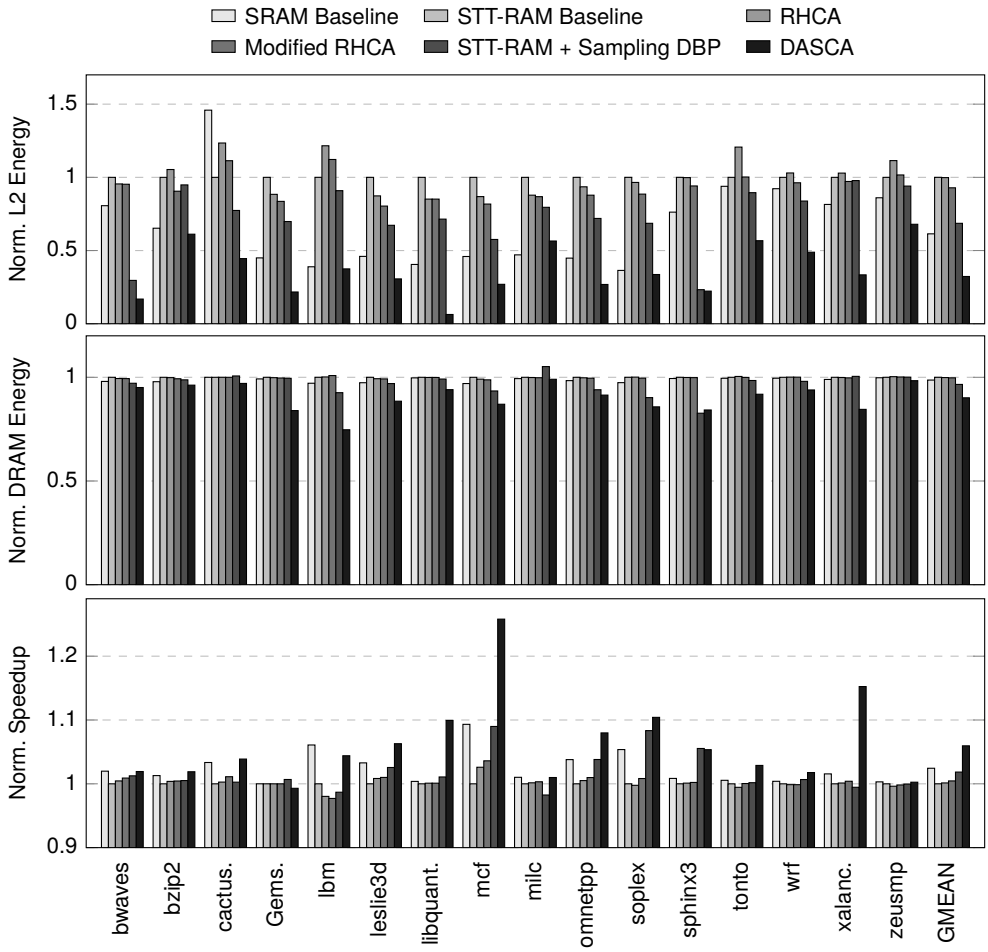


Figure 6.7: Energy consumption and speedup of the single-core system under different configurations.

by block fills and 74% of write energy consumed by writebacks from the L1 cache. On the contrary, the modified RHCA reduces the former by 19% and the latter by 29%. Between the two, ours is more preferable for last-level caches where block fills are the major cause of write energy consumption as discussed in Section 6.1.1.

Furthermore, DASCA also reduces the energy consumption of the main memory by 10% on average compared to the STT-RAM baseline. This improvement can be explained by two reasons. First, DASCA improves the row buffer hit rates by 11% on average. This is because bypassing closing writes has an effect of performing writebacks earlier, which increases the locality between read and write commands to the main memory [109]. Second, DASCA reduces the number of main memory accesses by 3% on average since (1) bypassing dead-on-arrival fills increases the effective capacity of the L2 cache by not placing dead blocks and (2) bypassing closing writes also improves the cache efficiency by proactively evicting dead blocks from the L2 cache.

Finally, it improves the overall performance by 6% on average compared to the STT-RAM baseline. This can be explained by a combination of fewer write operations to the L2 cache, reduced main memory activities, and increased row buffer hit rates as discussed earlier. Among them, the last two contribute the most considering that our scheme achieves 4.2% performance improvement even under the assumption of symmetric read/write latencies of the STT-RAM cache.

On the contrary, we observed that RHCA is not actually effective in reducing energy consumption. Although RHCA and the modified RHCA reduce the dynamic energy consumption of the L2 cache by 11% and 19%, respectively, its total energy consumption is reduced by only 0.2% and 7% due to the high static power of SRAM. Such a problem is exacerbated on low intensity benchmarks (e.g., cactusADM, tonto, and zeusmp) where the portion of dynamic energy is relatively small due to infrequent write operations. Note that this problem does not belong to a specific hybrid cache architecture, but is applicable to any cache architectures that use SRAM caches along with STT-RAM caches to reduce write energy consumption.

Also, our architecture outperforms the existing dead block predictor, on which our dead write predictor is based, in all aspects. This is because dead block predictors eliminate dead-on-arrival fills only, which take 40% of dead writes on average (Figure 6.5). This shows the significance of the proposed classification of dead writes and our predictor design based on it in the context of energy-efficient STT-RAM caches.

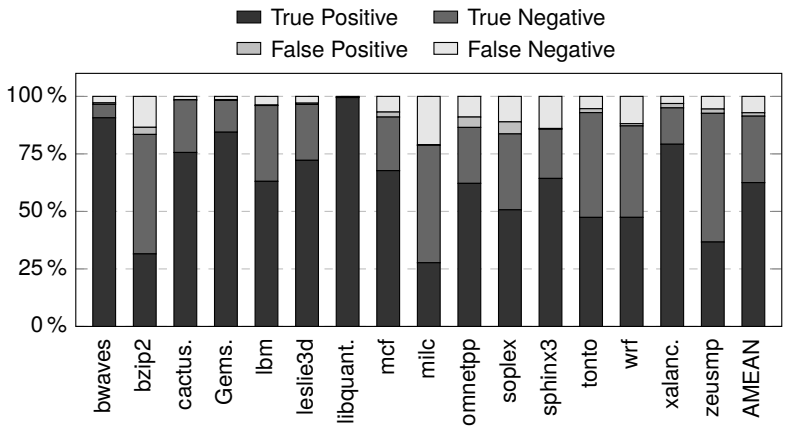


Figure 6.8: Coverage and accuracy of the dead write predictor.

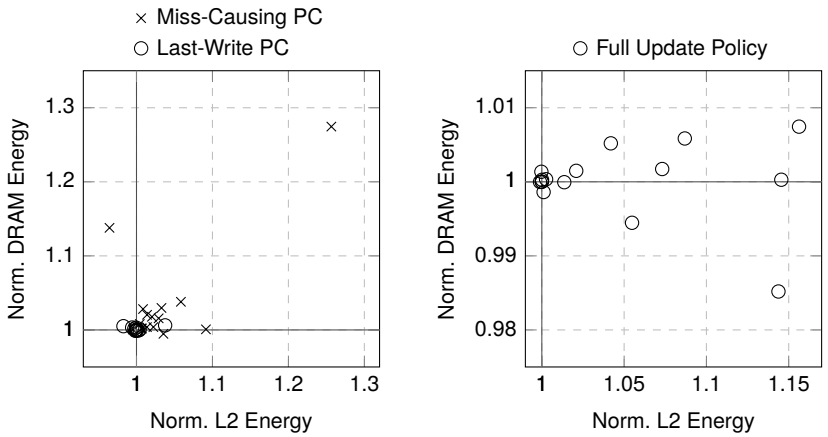
6.5.2 Coverage and Accuracy

One of the key factors that determine the effectiveness of our architecture is the coverage and accuracy of the dead write predictor. As long as the dead write prediction is accurate enough, our architecture can achieve a significant reduction in energy consumption of last-level caches and main memory. On the other hand, inaccurate prediction may not only miss such opportunity but also incur extra main memory accesses. In particular, the latter needs to be avoided as much as possible because one off-chip main memory access consumes orders of magnitude higher energy compared to one last-level cache access.

Figure 6.8 shows the coverage and accuracy of our dead write predictor. Two dark bars (true positive and true negative) represent correct predictions, whereas two light bars (false positive and false negative) represent mispredictions. On average, our mechanism achieves 91% of prediction accuracy. Most importantly, it minimizes the false positive errors (1.4% on average), which may incur extra main memory accesses. Although false positive errors reach up to 5% in some applications, our architecture is still able to reduce the energy consumption of the L2 cache and the main memory due to the improvement in row buffer hit rates and cache efficiency as discussed earlier.

6.5.3 Sensitivity to Signature

In order to accurately predict dead writes, it is important to choose a good signature that can represent dead writes. In this section, we compare the last-touch PCs, our proposed signature for writebacks, against two different types of signatures:



(a) Signatures (b) Update policy

Figure 6.9: Effect of using different types of signatures or update policies (normalized to our mechanism).

- Miss-causing PCs, in which instruction addresses that cause last-level cache accesses are used as signatures regardless of the type of write operations [88]. This type of signatures can be implemented without adding partial PC fields to lower-level cache tags unlike the other types of signatures.
- Last-write PCs, in which instruction addresses that write lower-level cache blocks for the last time are used as signatures for writebacks [109].

Figure 6.9a shows the energy consumption of the L2 cache and the main memory under different signatures, which are normalized to those for our signature. The results show that our signature outperforms both miss-causing PCs and last-write PCs on average. For accurate prediction of closing writes, signatures should be able to represent (1) whether the corresponding cache access is a write operation and (2) whether it is the last access to the block. While the miss-causing PCs contain neither of them and the last-write PCs have only the former,[5] our signature can represent both of them thereby improving the prediction accuracy for closing writes.

[5]Although the last-write PCs can represent whether the access is the last *write* to the block, it does not necessarily mean that it is the last access to the block, which is mandatory to be a closing write.

6.5.4 Sensitivity to Update Policy

Another important factor for dead write prediction is the policy to determine when to update the predictor table. Therefore, we compare our selective update policy, which does not update the predictor table on read hits, against the full update policy used in the baseline dead block predictor where every cache access updates the table.

Figure 6.9b compares the full update policy with our selective update policy. We observed that our policy identifies more dead writes than the full update policy does by preventing cache accesses that can never be dead writes from interfering with dead writes as discussed in Section 6.3.1. This facilitates the selective update policy to provide higher energy reduction compared to the full update policy.

6.5.5 Implications of Device-/Circuit-Level Techniques for Write Energy Reduction

One of the most common approaches to write energy reduction is to add circuitry that eliminates redundant bit-writes, e.g., differential writes [34, 49, 110] and Flip-N-Write [70]. Although our baseline does not adopt such techniques due to the overhead of additional circuitry, our scheme is orthogonal to them and is still very effective even if the baseline adopts them. According to our evaluation, DASCA still reduces the energy consumption of the L2 cache by 53% on average even if both the STT-RAM baseline and our scheme adopt differential writes.

Also, device-level innovations could potentially diminish the problem of high write energy in future, such as perpendicular cells [12] and shorter retention cells [43–45] with improved reliability. However, even if such technologies overcome many challenges that they are currently facing and eventually become feasible, our architecture is still expected to play an important role in reducing the energy consumption of last-level caches. Our evaluation shows that DASCA still achieves a 28% energy reduction in the L2 cache compared to the STT-RAM baseline under an aggressive projection where such technologies could eventually reduce write energy down to the level of read energy (89% reduction in write energy). As discussed in Section 1.2.1, however, the asymmetry between read and write energy is expected to remain in future due to the intrinsic limitation of the STT-RAM cell structure.

6.5.6 Impact of Prefetching

DASCA is perfectly interoperable with any type of prefetchers when prefetch buffers [111] are used to prevent cache pollution due to useless prefetches. This is because prefetched

blocks are placed on prefetch buffers first and are loaded into the cache only on demand accesses, which contain PC information that can be used in our prediction scheme. Experimental results show that, if the baseline is equipped with a stream prefetcher (32 streams, prefetch distance of 32, prefetch degree of 4) [112] and a 32 KB, 8-way set-associative prefetch buffer, applying our scheme reduces energy consumption of the L2 cache (excluding the prefetch buffer) and the main memory by 68% and 6.5%, respectively, with 4.7% performance improvement. Note that the lower energy reduction in the main memory (10% vs. 6.5%) is simply because useless prefetches incur extra main memory accesses, which in turn reduces the portion of demand main memory accesses.

Although the absence of prefetch buffers currently limits the choice of prefetchers under our scheme to PC-directed ones,[6] we believe that using prefetch buffers is essential for prefetching in STT-RAM caches because they prevent useless prefetches from dissipating write energy. Nevertheless, it would be interesting to investigate the possibility of coordinating prefetchers without prefetch buffers and our dead write predictor for write energy reduction, which is our future work.

6.6 Evaluation for Multi-Core Systems

In this section, we evaluate our architecture on a quad-core system against the modified version of RHCA, an STT-RAM cache with Sampling Dead Block Predictor, and Obstruction-Aware Cache Management Policy (OAP) [113]. OAP improves the performance of STT-RAM last-level caches by identifying specific applications (or processes) that do not get any performance benefit from last-level caches and skipping all write requests from such applications for a certain period, which alleviates bank contention caused by long write latency. We use 0.1 million and 10 million cycles as the sampling period and the launching period, respectively, as in the original paper.

6.6.1 Energy Consumption and Speedup

Figure 6.10 compares the energy consumption and speedup of our architecture against various techniques on a quad-core system. To evaluate system performance for multi-programmed workloads, we use the weighted speedup metric [105], in which IPC of each application is normalized to its IPC obtained by running it alone on the STT-RAM baseline.

[6] Prefetch requests generated by other types of prefetchers might not have PC information, which is needed for dead write prediction.

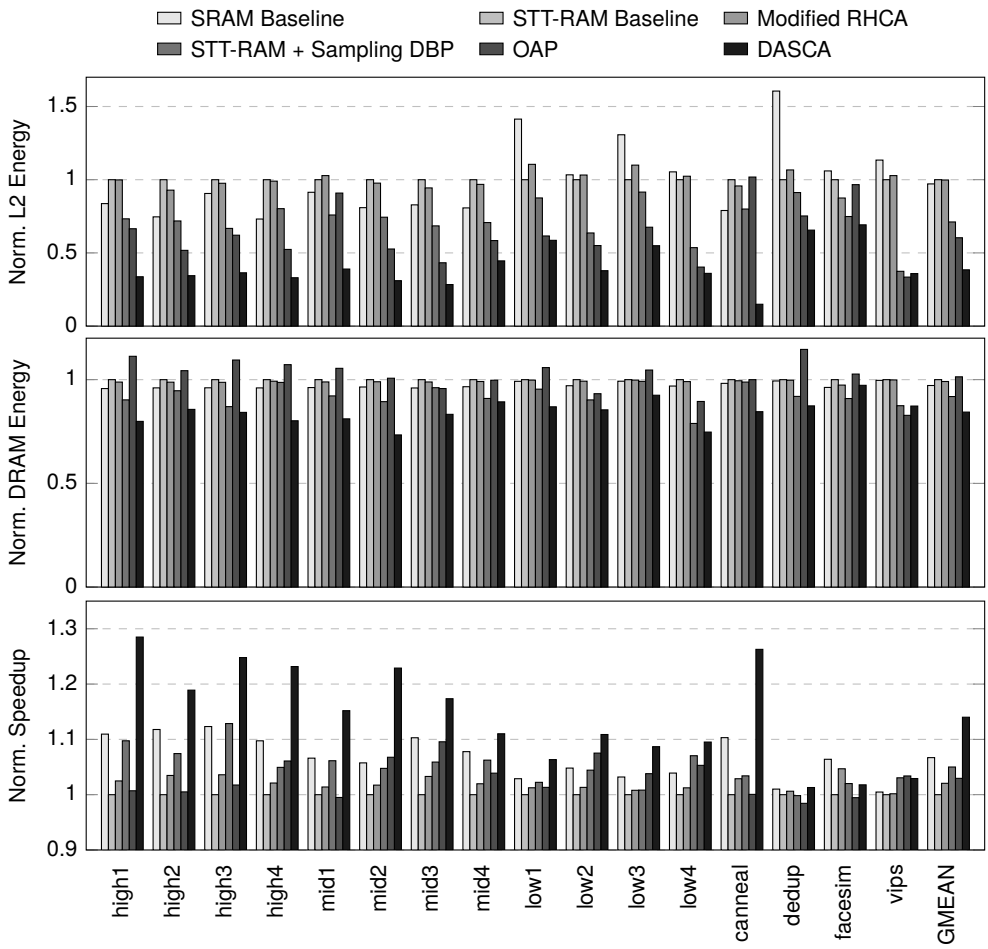


Figure 6.10: Energy consumption and speedup of the quad-core system under different configurations.

As shown in the figure, the SRAM baseline consumes almost the same energy as that of the STT-RAM baseline in contrast with the case of the single-core system (shown in Figure 6.7). The reasons of this are twofold. First, the quadrupled cache capacity increases the static power consumption of the L2 cache almost linearly, whereas its dynamic energy grows at a much slower rate. Second, as cache capacity gets larger, the smaller cell size of STT-RAM makes read operations even more efficient than those of SRAM. This trend suggests the need for STT-RAM last-level caches in future energy-efficient multi-/many-core systems, which are expected to have large last-level caches to provide higher on-chip memory bandwidth.

On top of that, DASCA further optimizes the efficiency of STT-RAM last-level caches, reducing the energy consumption of the L2 cache and the main memory by 62% and 16%, respectively, and achieving a 14% speedup on average over the STT-RAM baseline. Interestingly, the quad-core DASCA achieves higher energy reduction in the main memory and higher speedup compared to the single-core DASCA. This is because the increased number of cores puts more stress on the memory hierarchy, especially for the shared last-level cache and the main memory, and thus reducing accesses to them is much more influential in the case of the quad-core system.

Also, we observed that the existing solutions are not as efficient as our architecture in terms of both energy consumption and performance. In particular, OAP increases the energy consumption of the main memory in many benchmarks (11 out of 16). This comes from the fact that DASCA judiciously makes bypass decisions through *per-block* dead write prediction (which does not increase miss rates in an ideal case), whereas OAP decides it on a *per-application* basis and is unaware of block lifetime, thereby incurring a notable amount of extra main memory accesses. Considering that one main memory access consumes orders of magnitude higher energy than one last-level cache access does, such shortcomings can easily offset the energy reduction in the last-level cache, especially for memory-intensive workloads. Use of the existing dead block predictor alleviates the problem to some extent, though it is not as effective as ours because it is limited to eliminating dead-on-arrival fills only as discussed in Section 6.5.1.

6.6.2 Application to Inclusive Caches

Figure 6.11 shows energy consumption when the inclusion property is explicitly maintained between the L1 data caches and the L2 cache. It compares bypass-based schemes only, in which an adverse impact of inclusion is expected. The results are normalized to the inclusive variant of the STT-RAM baseline. OAP uses our bidirectional bypass scheme since its bypassing mechanism does not support inclusive caches. We

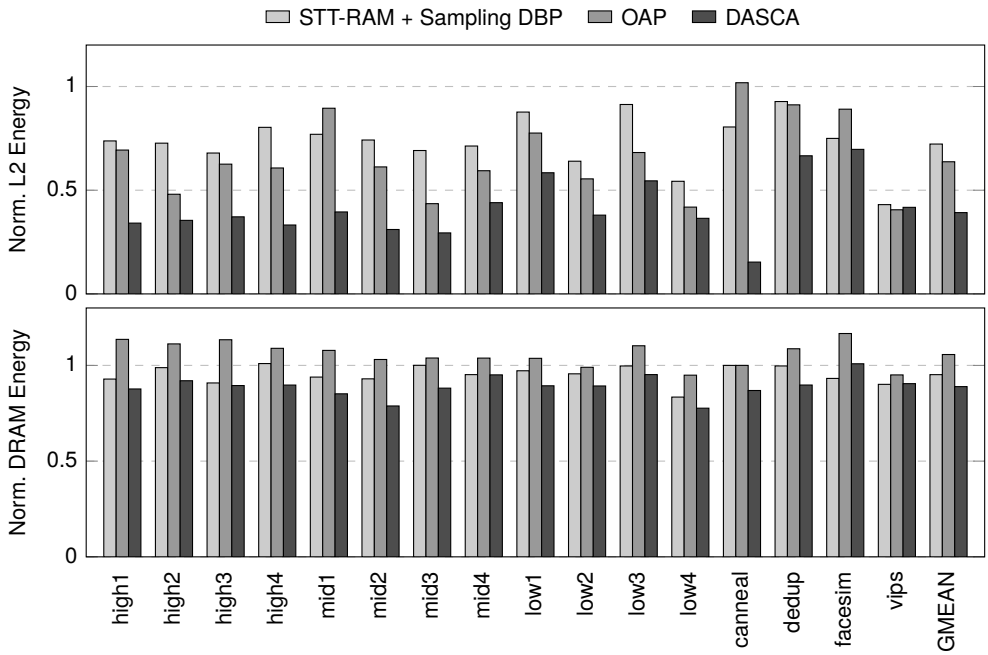


Figure 6.11: Energy consumption under a two-level inclusive cache hierarchy.

observed that the inclusion property negatively affects the energy consumption of the main memory in both schemes due to inclusion victims. However, while our architecture still achieves an 11% energy reduction in the main memory on average, OAP rather increases it by 6%, which is large enough to offset the energy reduction in the L2 cache, due to its unawareness of per-block access patterns.

6.6.3 Application to Three-Level Cache Hierarchy

Figure 6.12 compares the energy consumption of bypass-based schemes under a three-level cache hierarchy, in which per-core, 256 KB, 8-way set-associative caches are added between the L1 data caches and the L2 cache of our quad-core system. We found that all three schemes perform better in three-level caches than in two-level ones. This is because deeper cache hierarchies make last-level cache blocks less reused. Among the three schemes, DASCA outperforms the others in terms of total energy consumption, achieving 60% and 14% energy reductions in the L3 cache and the main memory, respectively. Although OAP achieves the same level of energy reduction in the L3 cache compared to our scheme, it comes with a cost of 9% higher energy consumption of the main memory on average.

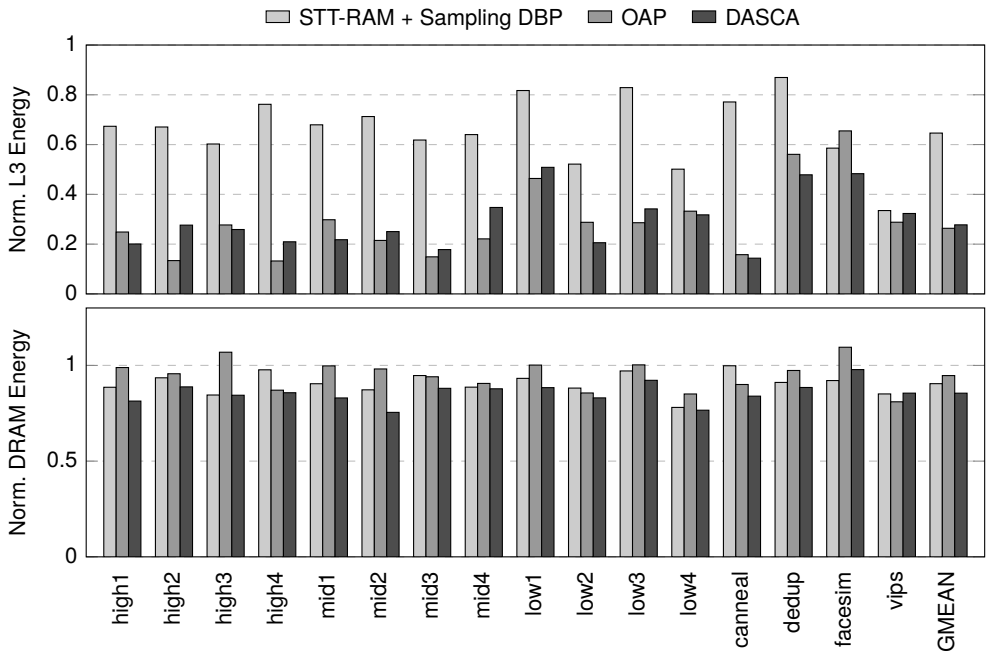


Figure 6.12: Energy consumption under a three-level cache hierarchy normalized to the STT-RAM baseline.

6.7 Summary

We proposed Dead Write Prediction Assisted STT-RAM Cache Architecture (DASCA), an architectural technique that effectively reduces high write energy consumption of STT-RAM last-level caches. DASCA is based on the key observation that a substantial amount of write operations can bypass last-level caches without any extra cache misses, which we call dead writes. In order to accurately predict dead writes, we designed a dead write predictor with our new signatures and update mechanism. We also introduced the bidirectional bypass scheme, which can skip both block fills and writebacks from lower-level caches for write energy reduction. Our evaluations show that DASCA significantly reduces the energy consumption of both last-level caches and main memory and achieves a notable speedup with modest storage overhead. It can also be used with many other techniques for further improvement as it does not require any modification to last-level caches.

Chapter 7

Link Power Management for Hybrid Memory Cubes

In the second part of this dissertation, we present architectures for *intelligent* main memory. As the first step, this chapter explores the potential of implementing advanced link power management and prefetching techniques inside main memory by leveraging the existing logic die of 3D-stacked DRAM. This work is motivated by our observation that, while HMCs provide much higher off-chip memory bandwidth to the CPU compared to conventional DDRx SDRAM, the off-chip I/O interface of HMCs consumes a significant amount of power *even when it does not transmit any data*. Although it is evident that the high bandwidth of HMCs could be a key to mitigating the memory wall problem, such power dissipation may not be worthwhile especially when the links are underutilized.

In this chapter, we propose an adaptive mechanism to partially disable[1] off-chip links of HMCs for energy reduction. To minimize the slowdown incurred by it, our scheme keeps track of the performance sensitivity to the number of enabled links and determines the number of links to be transitioned into the sleep mode based on it. In

This chapter is originally published in Proceedings of the Design Automation Conference, 2014 [114] and IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016 [115].

[1]In this work, 'disabling off-chip links' stands for setting the active links into low power states as explained in Section 7.1.2. Similarly, 'enabling links' is used as the opposite meaning of it (i.e., setting disabled links back to the active state).

order to disable as many links as possible, we try to minimize off-chip link activities. In particular, we focus on prefetching since it has been known to generate a significant amount of extra off-chip traffic due to useless prefetches [112]. Hence, we present two-level prefetching to reduce unnecessary off-chip link traffic in the presence of on-chip prefetchers. The key idea is to use two prefetchers with different aggressiveness and let the aggressive one prefetch blocks into an in-HMC prefetch buffer while the conservative one brings prefetch blocks into on-chip caches as usual.

7.1 Background and Motivation

7.1.1 Hybrid Memory Cube

As briefly introduced in Chapter 1, an HMC consists of one logic layer and multiple memory layers (see Figure 1.2). Each layer is composed of 16 or 32 partitions where vertically adjacent partitions constitute a *vault*. A vault is comprised of up to 16 DRAM banks, leading to a maximum of 512 banks per device. A vault has its own memory controller called vault controller, which schedules memory accesses and issues low-level DRAM commands to serve them. Vault controllers and DRAM banks are connected with through-silicon via (TSV) technologies for high-bandwidth and energy-efficient communication inside an HMC.

The I/O interface that connects an HMC with processors or other HMCs is based on high-speed serial links providing up to 320 GB/s of external bandwidth. In total, there can be up to eight full-duplex links for each device. A link is composed of either 8 or 16 lanes, each of which transfers one bit at a time with 10 to 15 Gb/s of the signaling rate. Between off-chip link transceivers and vault controllers, there exists a crossbar network to route each packet to the vault corresponding to its target address.[2] In other words, unlike traditional memory organization where each memory address is statically mapped to a specific channel, any link can be used to transmit a packet regardless of its target address. This facilitates more even distribution of loads across multiple links thereby improving link utilization.

Since HMCs are an emerging memory technology, little has been studied on their impact on performance and energy efficiency from the architectural perspective. Kim et al. [116] proposed a memory-centric network as a system interconnect, in which multiple sockets of processors communicate with each other through the inter-HMC network. Khurshid and Lipasti [117] utilized data compression techniques for thermal

[2] A packet can also be routed from one HMC to another if the current HMC does not contain the address.

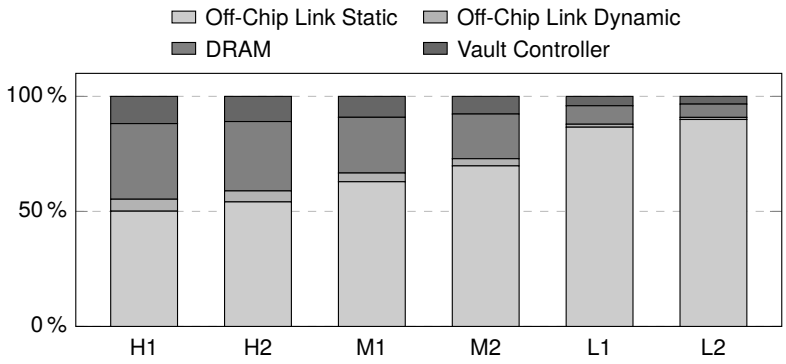


Figure 7.1: Energy breakdown of an HMC.

mitigation of HMCs thereby improving performance under temperature constraint. In this chapter, we focus on energy-efficient integration of HMCs through dynamic power management of off-chip links.

7.1.2 Motivation

Although high memory bandwidth of HMCs is one of their most promising characteristics from the performance perspective, it comes with several detrimental impacts on their energy efficiency. In particular, high-speed serial links consume a significant amount of power even when they are not used at all. This is mainly because the links consistently transmit null packets on idle in order to keep links aligned and ensure the recovered clock to be locked [26, 118]. Considering that such high power consumption of links may even dominate total power consumption of HMCs (e.g., 73% on average as shown in Figure 7.1, see Section 7.5.1 for detailed simulation configuration), mitigating its impact is of the utmost importance for better energy efficiency. To the best of our knowledge, there have been no studies so far that address this issue despite its importance.

One way of reducing power consumption of idle links is to set them into a low-power state. For this purpose, HMCs have a built-in feature for power state management of off-chip links with two low-power states, which are called sleep mode and down mode. Each link can enter into sleep mode on a per-link basis, wherein its high-speed SerDes circuitry is disabled and thus transmission of null packets on idle is avoided accordingly. The power consumption of a link can be further reduced by entering down mode, which also disables its phase-locked loop (PLL).

Partially disabling links is a promising solution especially for HMCs due to their unique organization. In traditional memory systems, disabling a channel precludes some

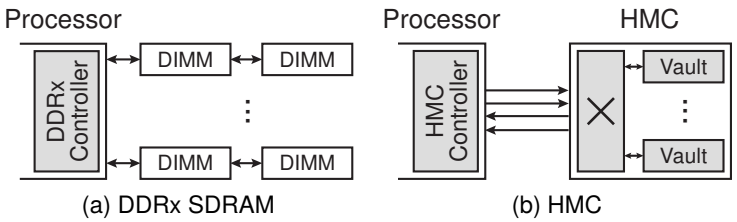


Figure 7.2: Simplified diagrams of link organization in DDRx and HMC.

memory addresses from being accessed because each memory address is statically mapped to a memory channel (see Figure 7.2a). On the contrary, there is no such limitation in HMCs due to the existence of an internal crossbar network that routes packets from any link to their target vaults (see Figure 7.2b). Thus, as long as each HMC has at least one enabled inbound/outbound link, all data stored in the HMC can be accessed without waiting for the disabled links to be enabled again.

Even though the connectivity of HMCs is not affected by disabling some of their links, links still need be disabled judiciously by considering its negative impacts on system performance due to the following reasons. First, disabling links reduces the external bandwidth of an HMC, which in turn may increase queueing delay of links. Our evaluation results shown in Figure 7.3 indicate that, although there is a great opportunity to disable several links while maintaining the same level of performance when the workloads are not memory-intensive (L1 and L2), it also has a possibility to incur huge performance degradation especially in the case of memory-intensive workloads (H1 and H2). Second, power state transition introduces long sleep and wakeup latency in the order of a few hundred nanoseconds and a few microseconds, respectively, which might delay memory accesses even further. The objective of this work is to fully utilize the capability of partially disabling off-chip links to reduce power consumption while minimizing its impact on performance.

Dynamic power management of links has been studied in the area of interconnection networks using on/off links [119–122]. However, most of them are not applicable for off-chip links of HMCs because of the following reasons. First, they assume that two nodes are connected with a single link, which imposes additional constraints to ensure connectivity of networks. This is obviously different from off-chip networks of HMCs where an HMC and a processor (or another HMC) are connected with multiple identical links. Also, they are usually designed to maintain throughput, whereas latency is also an important factor in main memory application of HMCs. Alonso et al. [121] did consider

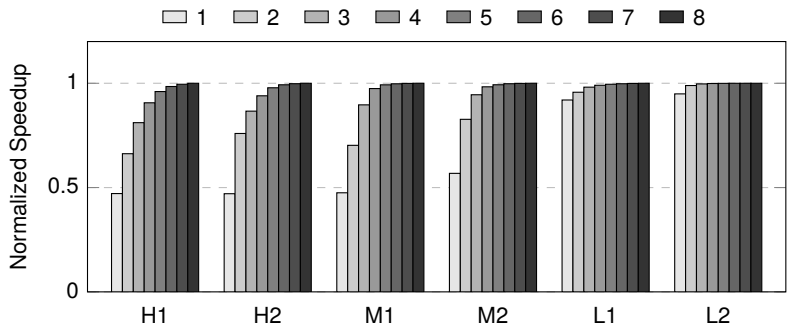


Figure 7.3: Normalized weighted speedup under different numbers of enabled links (e.g., '1' indicates one input link and one output link).

the former, but we will show that it provides a lower energy reduction compared to our mechanism (Section 7.5).

7.2 HMC Link Power Management

Our idea is to periodically estimate the minimum number of links required for negligible performance degradation with an average link delay as a proxy, and then enable or disable a link based on the estimation. The key observation behind it is that, when off-chip links are underutilized, disabling one of them has a negligible impact on the queueing delay of links. In the following sections, we explain our mechanism to determine the number of links to be disabled and perform power state transition. Note that our technique is not limited to off-chip links between a processor and an HMC, but can also be applied to links between two HMCs as will be discussed in Section 7.4.

7.2.1 Link Delay Monitor

Since the actual system performance under each link configuration (or the number of enabled links) is hard to estimate, we instead use an average link delay as a proxy for the system performance. The key question here is how to estimate *all average link delays, one for each possible link configuration, at the same time*.

For this purpose, we devise a hardware structure called *link delay monitor*. It simulates queueing delays of links under all possible link configurations from the requester side. For example, in a system where a processor is connected to an HMC, there should be two monitors, one in the processor for input links of the HMC and the other in the HMC for output links of the HMC.

A monitor consists of n saturating counters called *busy counters* for each possible link configuration having n enabled links, i.e., one counter for a single-link configuration, two counters for a dual-link configuration, and so forth. The purpose of the busy counters is to keep track of the remaining clock cycles for each (simulated) link to be idle. Thus, whenever a request arrives at the requester side, we choose a busy counter with the smallest value (representing the link that is the first to be available in future) for each configuration and increment each of them by the cumulative number of cycles to transmit the request packet over the link.[3] As time goes on, all busy counters are decremented by one for each cycle until it reaches zero.

A link delay monitor also has another set of counters called *delay counters*, one for each link configuration, to accumulate the sum of link delays (including queueing delays) of all requests. Note that the average link delay is obtained by dividing the sum of link delays by the number of requests. When a request arrives at the requester side, its link delay under each configuration can be calculated by a sum of (1) the number of cycles to transmit the request packet over the link and (2) the value of the selected busy counter (which represents the queueing delay). Thus, each delay counter is incremented by the sum of the two under the corresponding configuration.

Figure 7.4 illustrates an example sequence of requests that demonstrates the mechanism of updating the link delay monitor. Each set of rectangles represents a link delay monitor for a specific link configuration with n enabled links where a four-link device is assumed. Thick arrows marked as (A) to (D) indicate the time of request arrivals. Each request is assumed to take two cycles to be transferred across links. As explained previously, each time a request arrives, busy counters with the smallest value for each configuration are incremented by two (gray rectangles), whose values are then accumulated into delay counters. For example, at (C) in the 1-link configuration, the busy counter is incremented by two (i.e., $3 + 2 = 5$) and the delay counter is incremented by the value of the busy counter (i.e., $5 + (3 + 2) = 10$). All busy counters are decremented by one for each cycle. Note that busy counters contain the exact information about the actual link schedule.[4] For example, in the 3-link configuration, the value of the topmost rectangle at $t = 1$ (which is 1) indicates that the corresponding link will become idle at the next cycle, which is also true in the actual schedule for the same configuration. This implies that our scheme is able to simulate different numbers of enabled links

[3]This is defined by the burst length of the request and the signaling rate of links.

[4]We use the FCFS scheduling policy for the off-chip links of HMCs. The reason not to use other sophisticated scheduling policies is that vault controllers inside HMCs are capable of reordering memory accesses for better performance.

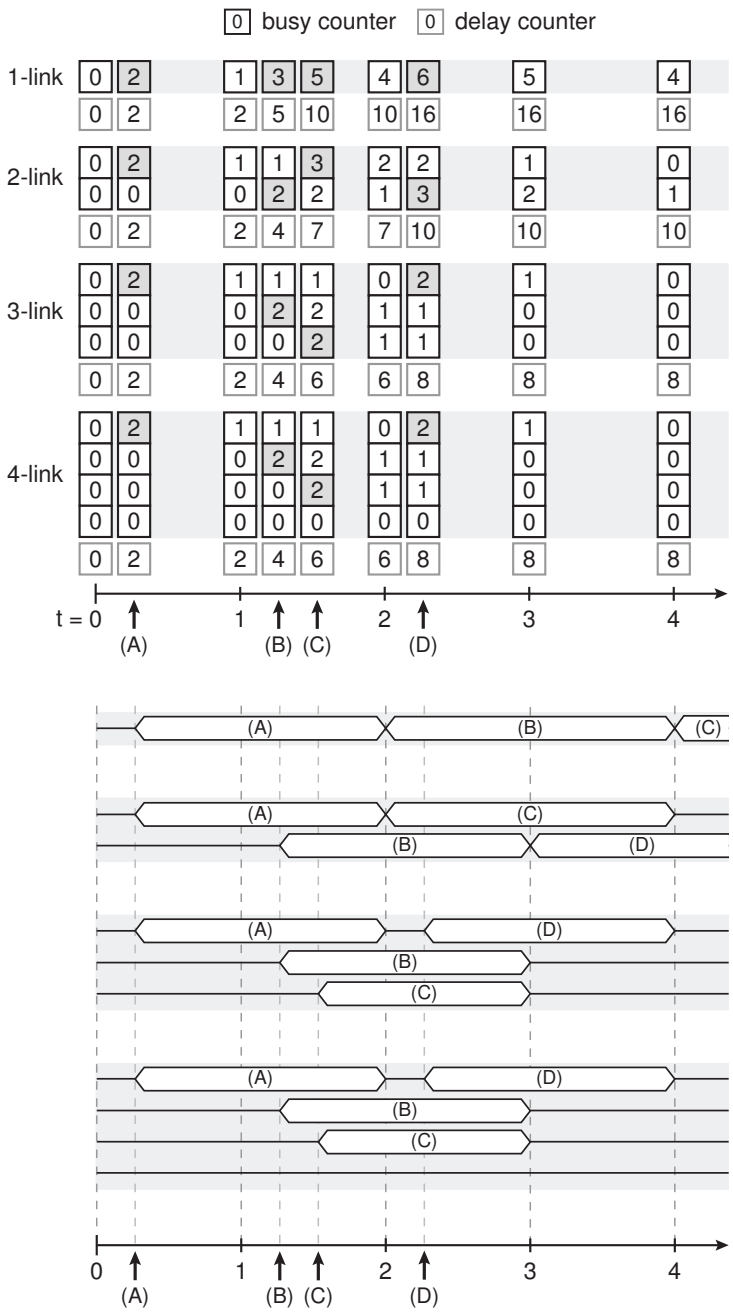


Figure 7.4: An example of updating the link delay monitor (top) and the actual link schedule (bottom).

at the same time with reasonable accuracy *without the need for actually trying every configuration one at a time*.

7.2.2 Power State Transition

The link delay monitor estimates the sum of link delays as described in the previous subsection. Based on this information, we formulate the problem of determining the number of links to be disabled as the problem of finding a configuration with the smallest number of links that satisfies the following constraint:

$$\frac{\overline{l_n} - \overline{l_N}}{\overline{m}} = \frac{\sum l_n - \sum l_N}{\sum m} \leq \alpha \cdot \frac{1}{u} \tag{7.1}$$

where l_n is a link delay under n enabled links ($1 \leq n \leq N$, N is the number of links per device), m is memory access latency, the sums of link delays are obtained from delay counters, u is link utilization, and α is a user-defined slowdown threshold. Although there can be many ways to estimate the sum of memory access latency m, we simply calculate it by multiplying the number of accesses with the row cycle time (tRC) of the DRAM as it does not need to be exact. This requires only one additional counter called *access counter* for each link delay monitor to keep track of the number of accesses.

The constraint can be interpreted as keeping the increase in link delays below $100\alpha/u\%$ of memory access latency on average. The rationale of introducing u into it is to consider the fact that applications with low memory intensity (low u) are less sensitive to an increase in link delays than those with high memory intensity (high u) are. In this work, we define $u = n_r/(n_r + n_n)$ where n_r and n_n are the numbers of transmitted real/null flits, respectively. Two additional counters are added for each link delay monitor to keep track of n_r and n_n.

From the perspective of a proxy for performance degradation, the aforementioned metric can be improved by considering types of memory accesses. To be more specific, system performance is known to be sensitive to memory read latency, whereas performance degradation due to increased memory write/prefetch latency can be hidden relatively easily. Therefore, we modify the constraint so that it monitors slowdown of link delays only for memory read accesses. This can be achieved simply by updating delay counters and access counters only on memory read requests and responses.

If the smallest number of links that satisfies the constraint in (7.1) is different from the current configuration, then power state transition is made. For this purpose, the link delay monitor drives power state transition pins of the target links from the requester side (note that link delay monitors are placed at the requester side as explained in the

previous section) to initiate power state transition of the links using the method stated in the HMC specification [26]. To adapt to dynamic behavior of applications, the power state transition of links is triggered periodically. Note that we allow only one link to be disabled or enabled for each period. This is not only to prevent too frequent power state transitions but also to avoid the situation where staggered power state transitions[5] take a nonnegligible portion of a transition period. At the end of each period, all counters except the busy counters are reset to start a new period.

The period of power state transition is chosen to be much longer than sleep and wakeup latency of links. Through this, the performance impact of the long transition latency can be minimized. Even during the transition time, requests can still be served by other enabled links due to the existence of in-HMC crossbar networks. In other words, long sleep and wakeup latency only affects the promptness of our scheme in adapting to application characteristics.

7.2.3 Overhead

Our scheme introduces a very low storage overhead. An 8-link device requires two link delay monitors, each of which includes 36 10-bit busy counters, eight 30-bit delay counters, and three 20-bit counters for constraint checks. Thus, the two monitors take only 165 bytes in total, which is negligible in terms of area as well as power consumption. Also, they are not on the critical path because updating counters can be done in background.

7.3 Two-Level Prefetching

Prefetching is a common optimization technique that reduces effective memory latency by proactively fetching data to be accessed in the near future. However, it inevitably consumes extra off-chip bandwidth because some of the prefetched blocks may not be accessed at all. According to Srinath et al. [112], an aggressive stream prefetcher improves the performance of single-core workloads by 84% on average at the cost of a 56% increase in off-chip bandwidth consumption. Although high memory bandwidth of HMCs may be sufficient to cope with such extra bandwidth consumption, it reduces the opportunity to disable as many links as possible through our power management technique by unnecessarily increasing off-chip link utilization.

[5] According to the HMC specification, simultaneous power state transitions are staggered by tSS in order to avoid supply voltage shifts [26].

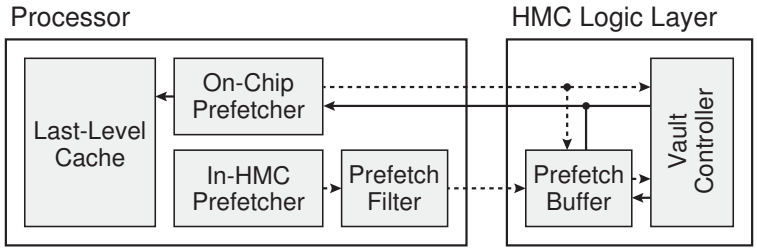


Figure 7.5: Two-level prefetcher organization (dashed/solid lines for requests/responses).

Therefore, we propose *two-level prefetching* to reduce off-chip link activities in the presence of prefetching while maintaining its performance improvement. In this design, the existing on-chip prefetcher is adjusted to be conservative (i.e., fewer prefetches)[6] in order to reduce unnecessary off-chip link traffic. Since this degrades system performance due to the increased average memory latency, we propose to add a new aggressive prefetcher called *in-HMC prefetcher*, which inserts prefetched blocks into a small SRAM buffer in HMCs called *in-HMC prefetch buffer*, to recover such performance degradation. This scheme has an effect of aggressively prefetching cache blocks within HMCs for memory latency reduction with low off-chip link bandwidth consumption (i.e., low link utilization), while judiciously bringing some of them to on-chip caches for link delay reduction.

Figure 7.5 shows the prefetcher organization of our approach. We separate the prefetch controller and the prefetch buffer of the in-HMC prefetcher. The in-HMC prefetch buffer is placed inside the HMC to minimize link activities, whereas the in-HMC prefetch controller is placed on the processor side because some prefetchers monitor both cache hits and misses for prefetch decision [107, 112, 123]. The in-HMC prefetch controller sends a prefetch request packet[7] to the target HMC, which lets the HMC prefetch the target cache block into its own in-HMC prefetch buffer. The prefetch buffer is managed in a manner similar to the previous work [111] where blocks are invalidated when they are either accessed or written back.[8] Note that low-cost integration

[6]The aggressiveness of prefetchers can be adjusted by using either different types of prefetchers or the same type of prefetcher with different levels of aggressiveness. For example, stream prefetchers allow controlling the aggressiveness by adjusting their prefetch distance and prefetch degree [112].

[7]A single prefetch request packet may contain multiple prefetch requests for consecutive cache blocks (e.g., stream prefetchers with prefetch degree greater than one).

[8]Although this requires every memory read/write to check the prefetch buffer to see if the target cache block is in the prefetch buffer, its energy cost is negligible due to its small capacity. More specifically, under our configuration shown in Section 7.5.1, a single prefetch buffer check consumes only 0.17% of DRAM read energy (assuming row buffer miss).

of in-HMC prefetch buffers is possible by utilizing the logic layer of HMCs without any modification of memory layers, which was not possible in traditional memory systems.

The downside of such remote prefetching is that it introduces a possibility to unnecessarily increase link utilization by sending prefetch requests even if the target cache blocks are already in the prefetch buffer. To prevent this situation, we add a prefetch filter on the processor side that keeps track of cache block addresses stored in the prefetch buffer. Each in-HMC prefetch request installs an entry for its target address into the prefetch filter, which is invalidated when the corresponding block is loaded into the last-level cache. If the target address is already in the prefetch filter, the corresponding in-HMC prefetch request is canceled to avoid unnecessary link traffic. Note that the information provided by the prefetch filter does not need to be exact because prefetching does not affect program behavior. In this work, we construct the prefetch filter with a tag array structure having the same number of sets and associativity as those of the prefetch buffer.

Our two-level prefetching scheme can be viewed as a combined form of processor-side prefetching [107, 111, 112, 123] and memory-side prefetching [52, 124, 125]. However, previous solutions based on memory-side prefetching assume either on-chip prefetch buffers (which do not reduce off-chip link traffic) or costly integration of prefetch buffers inside memory. On the other hand, our proposal is based on practical, low-cost integration of in-memory prefetch buffers by exploiting logic layers of HMCs. Moreover, our work is the first to study the synergistic effect between link power management and in-memory prefetching.

7.4 Application to Multi-HMC Systems

In order to scale the capacity of HMC-based main memory, the HMC specification [26] supports chaining multiple HMCs with high-speed serial links. For example, the simplest way of connecting multiple HMCs is a daisy-chain topology (shown in Figure 7.6) where HMCs are connected serially with each other. In such a case, links between two HMCs are used to transmit packets from/to the HMCs that do not have direct connection with the processor.

Our link power management scheme can be extended to such multi-HMC systems without any modification to the scheme. In order to employ our scheme to links between two HMCs, all we have to do is to add two link delay monitors between two HMCs just as in the links between the processor and an HMC. Although evaluations in this work consider the daisy-chain topology only, our scheme can be applied to any topology as long as two modules (HMCs or processors) are connected with at least two links

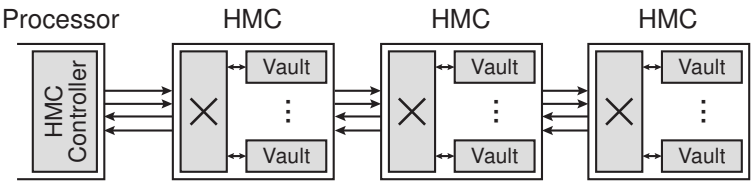


Figure 7.6: A system with multiple HMCs connected with the daisy-chain topology.

since it dynamically adapts to link activity with no assumption of specific topologies through link delay monitors. Also, we expect that the efficiency of our link power management scheme could be further improved by coordinated control of it with other system components (e.g., routing algorithms, topologies, page allocation, etc.), which will be explored in our future work. In fact, a good example of it is our two-level prefetching where link power management and on-chip prefetching are coordinated together.

Also, two-level prefetching can be easily applied to such systems as follows. First, the size of the prefetch filter is scaled linearly with the number of HMCs. This is because, assuming that each HMC has a prefetch buffer with the same size, the total capacity of prefetch buffers in the system increases linearly with the number of HMCs. Second, when an in-HMC prefetcher issues a prefetch, it sends a prefetch request packet to the target HMC that contains the target cache blocks to bring them to the in-HMC prefetch buffer. If the target cache blocks are stored in multiple HMCs (one prefetch request can initiate prefetches to multiple consecutive cache blocks as discussed in footnote 7), one packet is sent for each of the target HMCs. Note that multi-HMC systems still have only one in-HMC prefetcher, which is located at the processor side as shown in Figure 7.5.

7.5 Experiments

7.5.1 Methodology

We evaluate our technique using a cycle-accurate x86-64 simulator whose frontend is Pin [95]. Table 7.1 shows the details of the target system. Our baseline is equipped with a stream prefetcher, which improves the system performance by 17% compared to the system without prefetchers. Characteristics of DRAM layers and vault controllers are modeled by CACTI-3DD [126] and McPAT [127], respectively. In addition, prefetch buffers and prefetch filters are modeled by using ITRS LSTP devices. All models are based on a 32 nm technology node.

Table 7.1: Simulation Configuration

Component	Configuration
Processor	8 cores, 3 GHz, out-of-order, 4-issue superscalar, 64-entry instruction window
L1 I/D Cache	private, 32 KB, 4-way, 64-byte blocks, 16 MSHRs
L2 Cache	shared, 4 MB, 16-way, 64-byte blocks, 64 MSHRs
Prefetcher	Stream prefetcher with 64 streams [112] Baseline: PF distance of 64, PF degree of 4 Two-level: PF distance of 8/64, PF degree of 1/4 (on-chip/in-HMC)
PF Buffer	16 KB per vault, 16-way, 64-byte blocks
PF Filter	16-way, 512-set tag array
Off-chip	8 full-duplex links (8 input and 8 output links),
Link	8 10 Gb/s lanes per link (half-width configuration [26])
LPM	Period: 100 µs, slowdown threshold α (see (7.1)): 0.05
HMC	8 DRAM layers, 32 vaults, 2 banks per vault, 1 KB pages, line-interleaved address mapping
Vault	32-entry command queue
Controller	FR-FCFS scheduling with the closed-row policy [97]

For dynamic power management of off-chip links, we use the sleep mode instead of the down mode as a low-power state since it has relatively short transition latency. The sleep latency of a link is set to 680 ns according to the specification [26], while its wakeup latency is conservatively assumed to be 5 µs. Energy consumption of a link is modeled as 2.0 pJ/bit (real packets) and 1.5 pJ/bit (null packets) by referring to the previous research [116]. We assume that the sleep mode reduces the energy consumption of links by 95% considering that a PLL takes approximately 5% of total power consumption of a high-speed serial link transceiver when it is shared across eight links [128, 129]. During the power state transition, links are assumed to consume the same amount of energy as if they are enabled. Since links cannot transmit data (but consume power) during power state transition, this models energy overhead for power state transition. Note that this overhead does not exceed 5% of link energy consumption even in the worst case because our scheme allows at most one power state transition per period (explained in Section 7.2.2) and one period is 20x longer than the maximum of the sleep latency and the wakeup latency.

We use 12 multiprogrammed workloads for our experiments, each of which is composed of eight SPEC CPU2006 [101] applications. To evaluate various applications

Table 7.2: Multiprogrammed Workloads

ID	Workloads
H1	gcc, lbm, lbm, leslie3d, libquantum, milc, soplex, sphinx3
H2	Gems., Gems., bwaves, lbm, leslie3d, leslie3d, omnetpp, sphinx3
H3	Gems., gcc, lbm, libquantum, mcf, omnetpp, omnetpp, sphinx3
H4	bwaves, gcc, lbm, libquantum, mcf, soplex, soplex, sphinx3
M1	Gems., lbm, libquantum, omnetpp, bzip2, bzip2, h264ref, zeusmp
M2	gcc, libquantum, milc, sphinx3, cactus., gromacs, wrf, zeusmp
M3	Gems., lbm, leslie3d, soplex, bzip2, hmmer, wrf, zeusmp
M4	bwaves, gcc, lbm, mcf, gromacs, h264ref, perlbench, perlbench
L1	astar, astar, bzip2, cactus., cactus., h264ref, perlbench, tonto
L2	astar, bzip2, cactus., gromacs, gromacs, hmmer, perlbench, tonto
L3	astar, cactus., cactus., gromacs, perlbench, tonto, zeusmp, zeusmp
L4	astar, cactus., cactus., hmmer, perlbench, tonto, tonto, wrf

with a wide range of memory intensity, we first choose 21 applications whose misses per kilo instructions (MPKI) under a 512 KB L2 cache is greater than one and classify them into two categories: 11 applications with MPKI $\geq$ 20 (denoted as HIGH) and 10 applications with $1 \leq$ MPKI $<$ 20 (denoted as LOW). Then, we randomly select eight HIGHS to compose a multiprogrammed workload with high memory intensity (H1 to H4), eight LOWS for a workload with low memory intensity (L1 to L4), and four HIGHS and four LOWS for a workload with medium memory intensity (M1 to M4). Table 7.2 shows the list of workloads used in this work. We run the simulation for 800 million instructions in total, where each application starts from the beginning of its representative phase [103].

7.5.2 Link Energy Consumption and Speedup

Figure 7.7 compares link energy consumption (including both static energy and dynamic energy) and weighted speedup[9] of various approaches. LPM and 2LP represent our link power management technique and two-level prefetching, respectively. Also, we include comparison with a power saving mechanism for on/off links of high-degree switches (denoted as HDS) [121], although it is not directly targeted for off-chip links of

[9]The weighted speedup [105] is computed by normalizing IPC of each application to its IPC measured by running it alone in the baseline with a 512 KB L2 cache and calculating the sum of them. This prevents from favoring high-IPC applications over low-IPC ones.

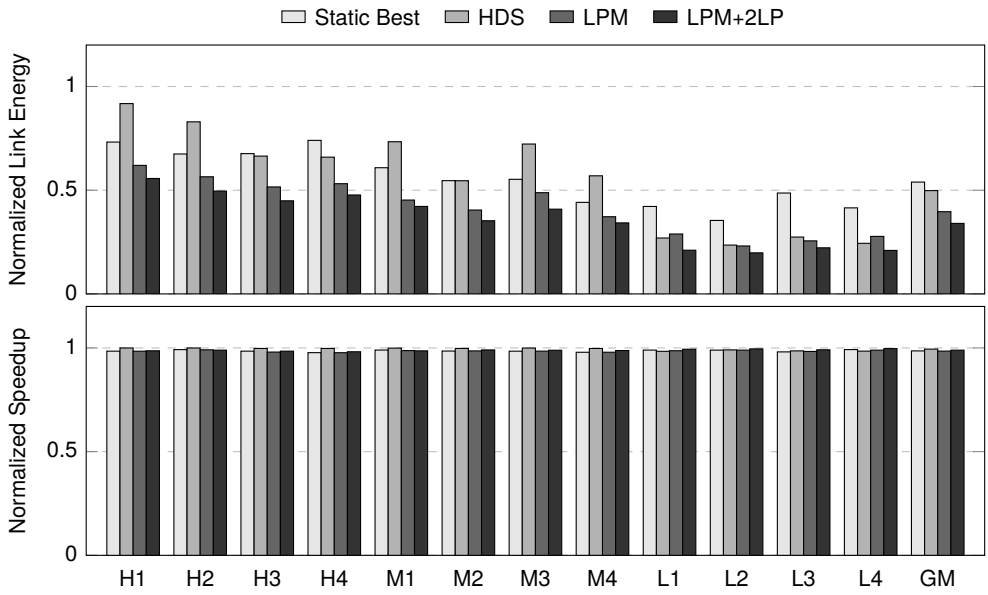


Figure 7.7: Comparison of link energy consumption (top) and weighted speedup (bottom), both of which are normalized to the system without link power management.

HMCs. We set its thresholds to $u_L = 0.15$ and $u_H = 0.3$ [121]. We also compare a static approach (denoted as Static Best) where each workload uses its best link configuration, which is determined by choosing the one with the performance closest to that of LPM among all 64 possible configurations through profiling. Note that this is impractical in many cases since such profiling information may not be available especially for multiprogrammed workloads (which require profiling all possible combinations of workloads and execution schedule). The last sets of bars labeled as 'GM' indicate geometric means across 12 workloads.

As shown in the figure, our mechanism for dynamic link power management achieves a 60% reduction in link energy consumption with 1.5% performance degradation. On average, our mechanism disables 74% of input links of the HMC and 55% of its output links. The reason why input links are disabled with a higher ratio is that they transmit a 16-byte request packet on a read request, which is much shorter than the corresponding 80-byte response packet transmitted through the output links. The important aspect of our approach is that such high energy reduction is achieved without any notable performance degradation, even though the best number of disabled links varies across workloads as shown in Figure 7.3. This, together with the fact that our mechanism

outperforms the static approach by a large amount, clearly indicates the good adaptivity of our mechanism to both application characteristics and runtime variations.

Furthermore, our two-level prefetching provides a 5.6% additional link energy reduction with a slight improvement in performance (1.1% performance degradation compared to the baseline). The major reason of this improvement comes from the 14% reduction in off-chip link bandwidth consumption by preventing useless prefetches from being transferred through off-chip links. Although the in-HMC prefetcher generates additional off-chip traffic due to prefetch request packets, it contributes only 2.3% of the entire off-chip traffic on average.

Also, we observed that our techniques provide much higher energy reduction compared to the previous approach for high-degree switches, especially in workloads with high memory intensity. The major difference of our scheme against the previous approach is that, while the previous approach uses link utilization (or throughput) as the only metric, our scheme considers both throughput and latency in determining the number of links to be disabled.

Note that, although performance degradation incurred by our scheme could possibly increase the energy consumption of other components in the system due to longer execution time, its impact is expected to be negligible compared to the link energy reduction achieved by our architecture. This is because main memory is the single dominant source of total energy consumption in servers (e.g., 48–62% according to the estimation done by Volos et al. [130]), which will be the primary target for HMCs.

7.5.3 HMC Energy Consumption

Figure 7.8 shows the energy consumption of an HMC under the baseline and our architecture. On average, our approach reduces the energy consumption of the HMC by 52%. This signifies the importance of off-chip link power consumption in energy-efficient main memory systems using HMCs. In the case of the baseline, the major portion of the energy consumption is taken by off-chip links (73% on average). Our techniques reduce this portion down to 49% with negligible overhead from the prefetch buffer and the prefetch filter.

7.5.4 Runtime Behavior of LPM

Figure 7.9 shows how links are enabled or disabled over time through our link power management technique (LPM in Figure 7.7). The results imply that the number of links to be disabled not only depends on application characteristics (e.g., H1 vs. L1) but also varies over time (e.g., around 10 ms vs. around 40 ms in M1). Importantly, Static

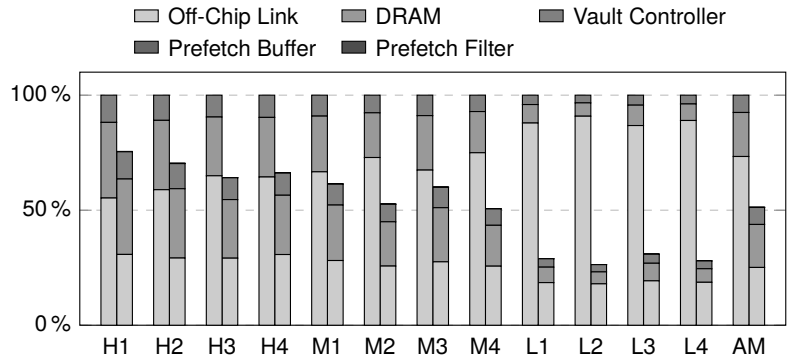


Figure 7.8: Energy consumption of an HMC under the baseline (left) and our approach (LPM+2LP, right).

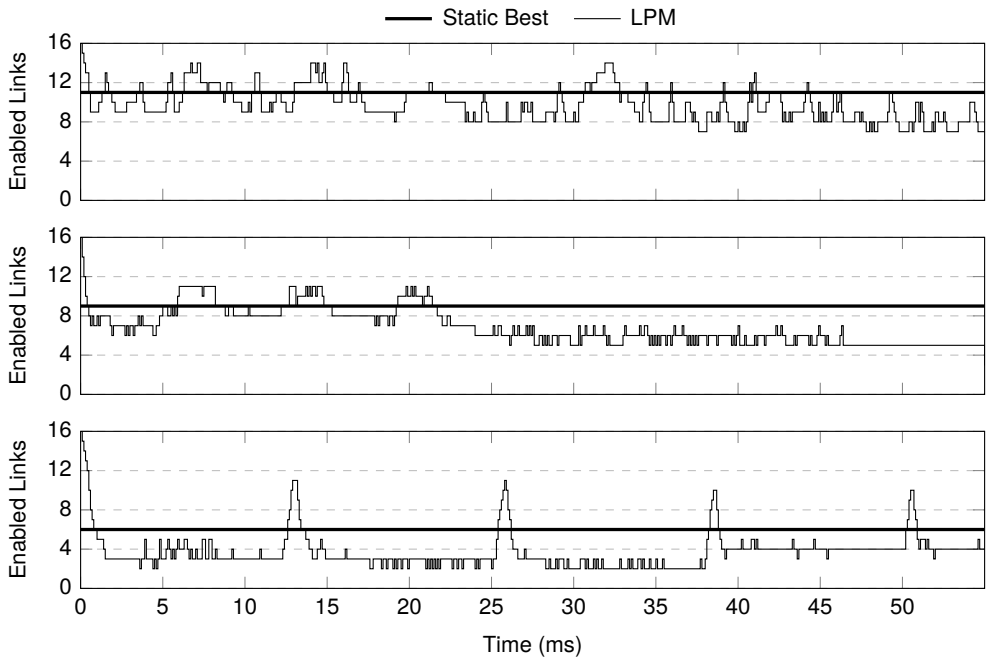


Figure 7.9: Changes in the number of enabled links over time in Static Best and LPM (top: H1, middle: M1, low: L1).

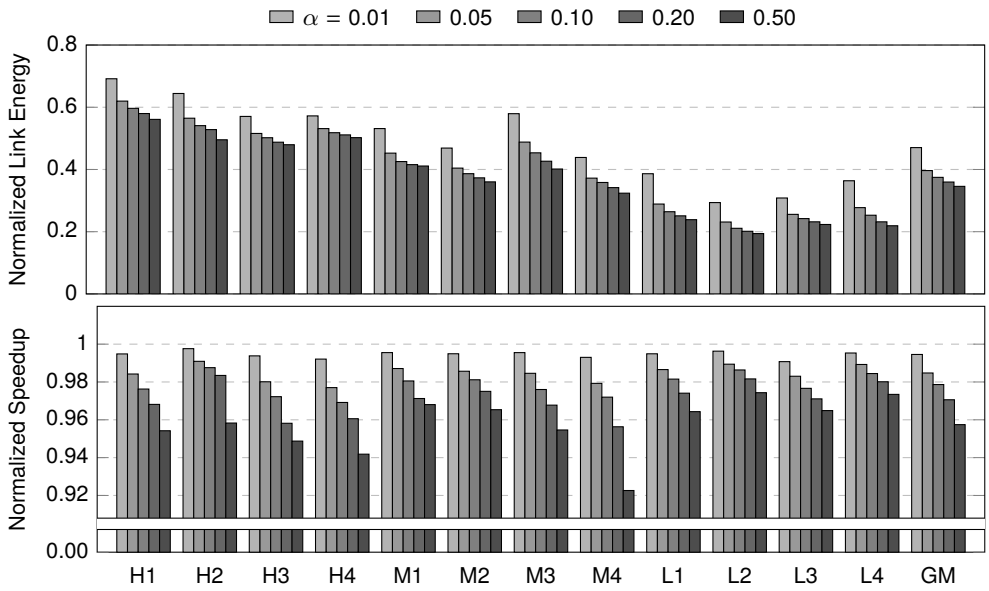


Figure 7.10: Impact of slowdown threshold α on link energy (top) and performance (bottom).

Best is not able to follow this behavior but averages it over the entire execution as shown in the figure, which explains why our techniques outperform Static Best by a large margin as shown in Figure 7.7. Also, this emphasizes the importance of dynamic hardware-based mechanisms for link power management since it is challenging for software-based and/or profiling-based approaches to adapt to time-varying behavior of applications in a fine-grained manner.

7.5.5 Sensitivity to Slowdown Threshold

Figure 7.10 compares normalized link energy and speedup while varying the slowdown threshold α. As expected in (7.1), higher slowdown thresholds achieve higher link energy reduction at the cost of performance degradation. This trade-off in our scheme provides freedom to tune α according to the maximum tolerable slowdown of the target system.

7.5.6 LPM without Prefetching

Figure 7.11 shows the experimental results when both the baseline and LPM do not employ prefetching. As mentioned before, the existence of prefetchers improves system

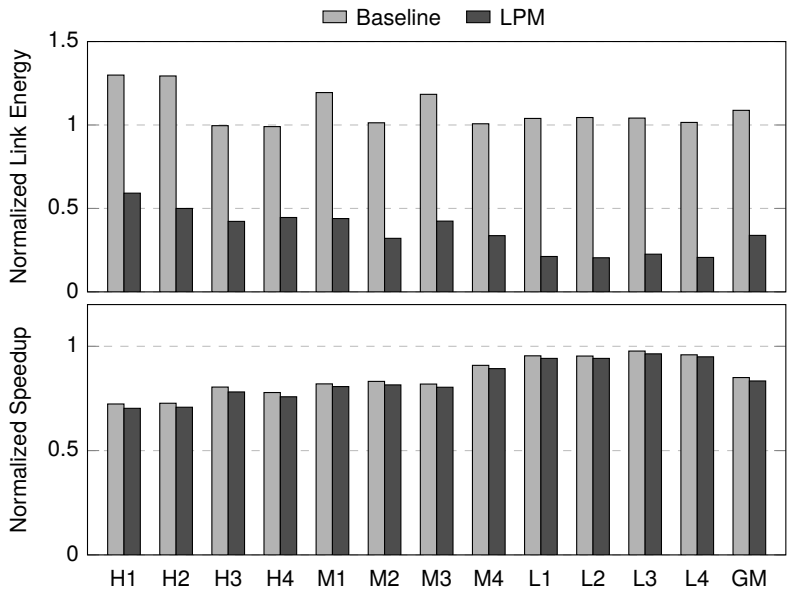


Figure 7.11: Evaluation of link energy (top) and speedup (bottom) under the system without prefetching (normalized to the baseline with prefetching).

performance by 18% compared to the system without it. Due to this, disabling the prefetcher in the baseline increases the link energy consumption by 8% on average since links consume more static energy without prefetching due to the longer application execution time. Also, we found that the effectiveness of our technique on link energy reduction becomes even higher without prefetching, i.e., 69% on average compared to the baseline without prefetching. This is because excluding prefetchers reduces on-chip link utilization thereby facilitating more links to be disabled.

7.5.7 Impact of Prefetching on Link Traffic

Figure 7.12 compares off-chip link traffic (total number of transmitted real flits) in the systems without prefetching, with the conventional on-chip prefetching, and with our two-level prefetching. Most importantly, on-chip prefetching significantly increases off-chip traffic (21% on average) due to incorrect prefetches. On the contrary, our two-level prefetching consumes almost the same off-chip bandwidth compared to the system without prefetching by keeping most of prefetched blocks inside HMCs. Such reduced off-chip traffic further improves the energy efficiency of our link power management scheme by allowing it to disable more links under the same performance constraint.

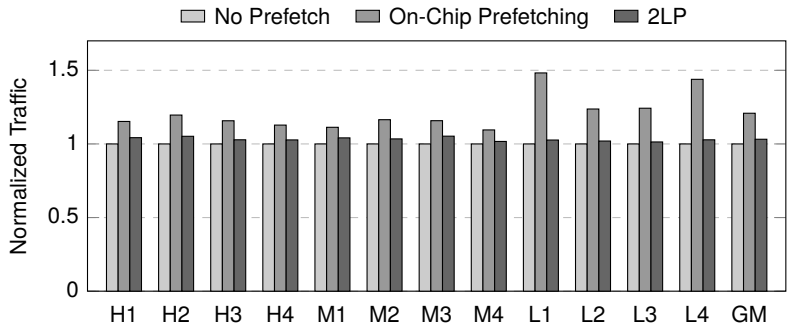


Figure 7.12: Normalized off-chip link traffic under various prefetching schemes.

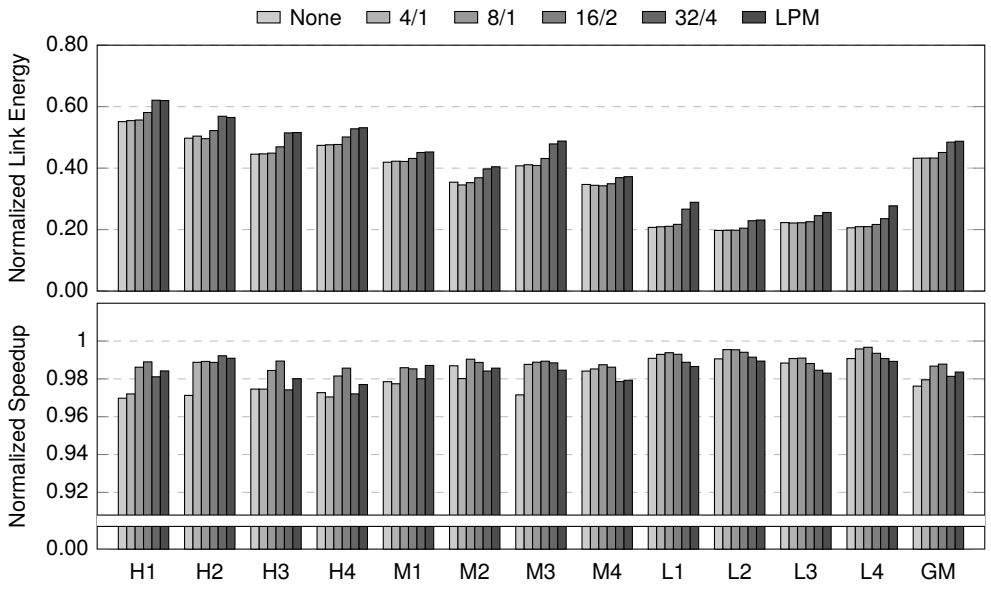


Figure 7.13: Impact of on-chip prefetcher aggressiveness in two-level prefetching on link energy (top) and speedup (bottom).

7.5.8 On-Chip Prefetcher Aggressiveness in 2LP

Figure 7.13 shows normalized link energy and speedup under two-level prefetching with different levels of on-chip prefetcher aggressiveness. Each configuration represents prefetch distance and prefetch degree of the on-chip prefetcher, e.g., '4/1' for prefetcher distance of 4 and prefetcher degree of 1. Longer prefetch distance and higher prefetch degree lead to more aggressive prefetching [112]. 'None' indicates the configuration with the off-chip prefetcher only (i.e., no on-chip prefetcher) and 'LPM' indicates the one with the on-chip aggressive prefetcher (64/4) only (i.e., no two-level prefetching).

According to the experimental results, two-level prefetching without on-chip prefetchers shows up to 2.0% of performance degradation compared to LPM. This is because it increases the average L2 cache access latency by up to 14%. On the other hand, aggressive on-chip prefetchers (e.g., 32/4) provide almost no benefit in link energy reduction compared to LPM because they also introduce extra off-chip link activities. We found that an on-chip prefetcher with prefetch distance of 8 and prefetch degree of 1 balances between these two aspects thereby achieving a 5.4% additional link energy reduction with 0.3% performance improvement compared with LPM. Since the effectiveness of on-chip prefetchers in two-level prefetching varies across applications, prefetching with adaptive aggressiveness [112, 123, 124] could also be adopted for on-chip prefetchers.

7.5.9 Tighter Off-Chip Bandwidth Margin

In our baseline configuration, eight full-duplex links provide 80 GB/s of off-chip bandwidth, which may not be easily saturated depending on application characteristics. Thus, in order to analyze the impact of tighter off-chip bandwidth margin, we evaluate our architecture with reducing the number of links in the baseline from eight to four. From the result shown in Figure 7.14, we made two interesting observations.

First, even with such tighter bandwidth margin where disabling only a few links could degrade performance significantly (see one- to four-link configuration in Figure 7.3), our link power management scheme minimizes the performance overhead to 0.7% while reducing the HMC energy consumption by 25% on average (without 2LP). This indicates that our scheme accurately estimates the number of links that can be disabled without a noticeable impact on system performance.

Second, our two-level prefetching not only achieves a further reduction in HMC energy consumption but also *improves* performance by up to 5.9% compared to the baseline (especially in memory-intensive workloads). This is because it reduces

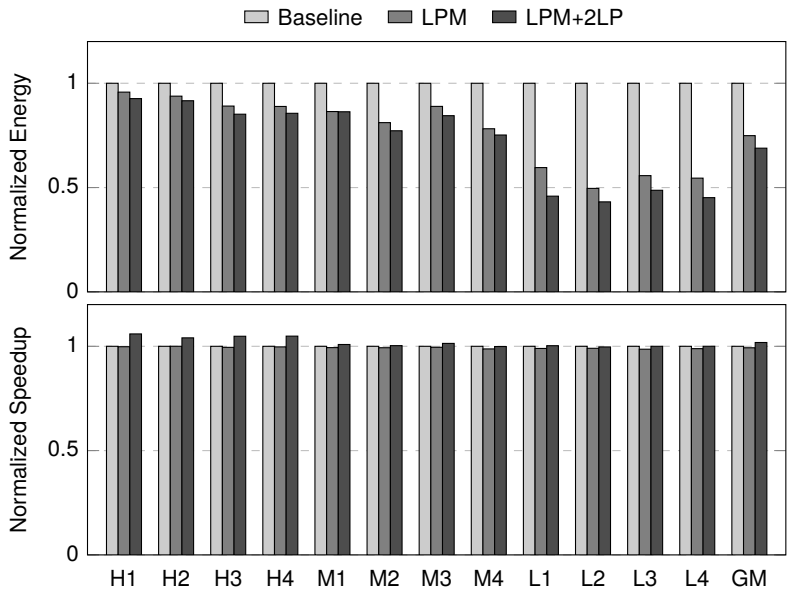


Figure 7.14: Evaluation of HMC energy consumption (top) and weighted speedup (bottom) under the system with four full-duplex links (instead of eight in the default configuration).

unnecessary off-chip bandwidth consumption caused by useless prefetches, which in turn shortens queueing delay of links under high link contention.

7.5.10 Multithreaded Workloads

In addition to multiprogrammed workloads used so far, Figure 7.15 shows evaluation results of our architecture for the PARSEC 2.1 benchmark [104]. All workloads use simlarge input sets and are run for two billion instructions from the beginning of Region-of-Interest (ROI) defined in the benchmark. We use instructions per cycle (IPC) as a performance metric for multithreaded workloads.

As shown in the figure, our scheme reduces the HMC energy consumption by 73% with 0.7% performance degradation on average. The reason why it shows higher energy reduction than in Figure 7.8 is that PARSEC benchmarks are less memory-intensive compared to the multiprogrammed workloads used in our evaluations (average L2 MPKI of the former and the latter is 5.0 and 25.0, respectively). Also, this diminishes the effectiveness of two-level prefetching as the amount of useless prefetch is small due to infrequent main memory accesses.

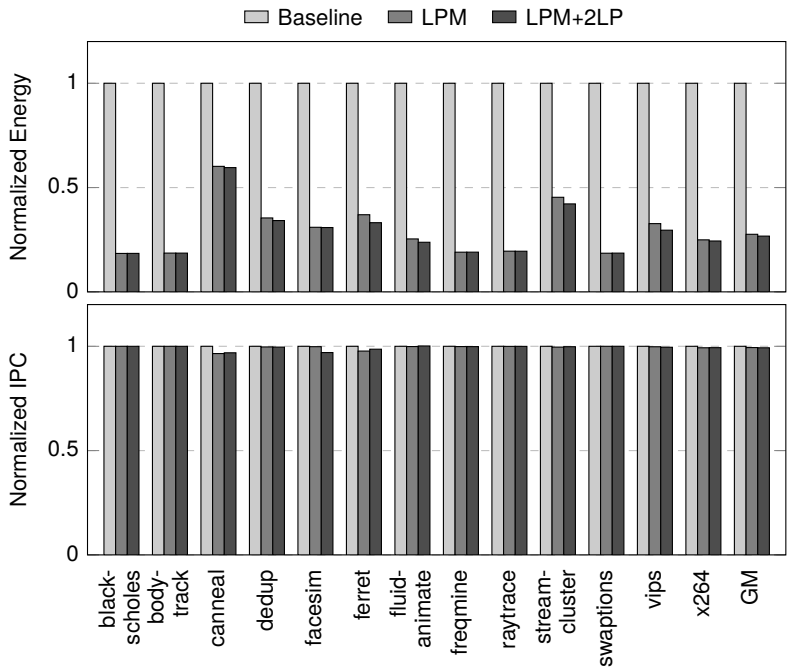


Figure 7.15: Evaluation of HMC energy consumption (top) and normalized IPC (bottom) using multithreaded workloads.

7.5.11 Multi-HMC Systems

We also evaluate our approach in systems equipping multiple HMCs as their main memory. We simulate a four-HMC system where four HMCs (each of which has the same configuration as the one shown in Table 7.1) are connected with a daisy-chain topology. The system is equipped with eight link delay monitors, i.e., two link delay monitors between two HMCs as well as between the host processor and an HMC. Since putting three more HMCs into our baseline system increases the total capacity of the in-HMC prefetch buffer by four, we quadruple the number of prefetch filter sets accordingly.

Figure 7.16 shows HMC energy consumption and performance of our architecture normalized to the baseline. Unlike Figure 7.7, we are not able to provide results for Static Best since determining the best link configuration for four sets of eight full-duplex links in the four-HMC system requires simulating too many combinations, or more precisely, $(8 \times 8)^4 = 16777216$ different link configurations for each workload.

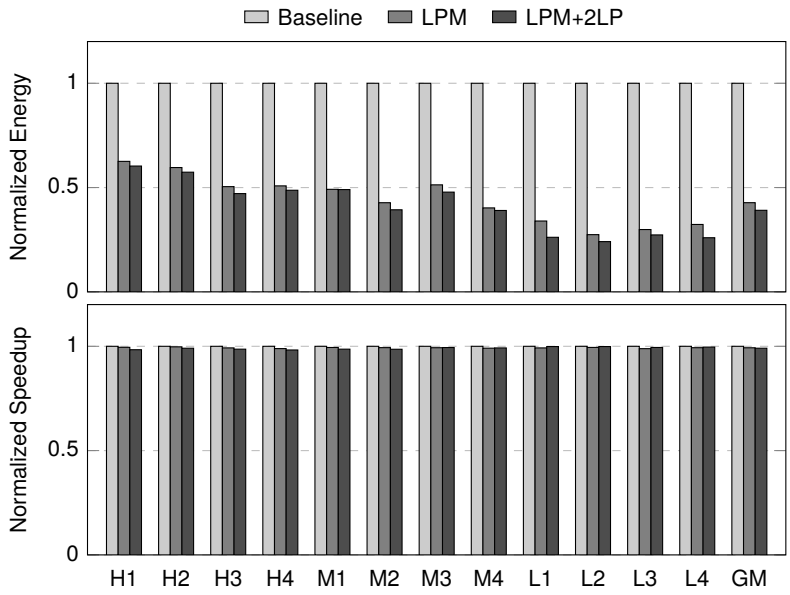


Figure 7.16: Effectiveness of the proposed technique in a system with four HMCs in terms of HMC energy consumption (top) and weighted speedup (bottom).

Compared to the results for the single-HMC system shown in Figure 7.8, our scheme achieves even higher reductions in HMC energy consumption (i.e., 52% in single-HMC systems vs. 61% in multi-HMC systems) with similar levels of performance overhead in the four-HMC system. This is because HMC-to-HMC links (which do not exist in single-HMC systems) show lower utilization compared to the host-to-HMC and HMC-to-host links.

In order to further analyze this, Figure 7.17 shows bandwidth utilization (i.e., the proportion of average bandwidth consumption to the maximum link bandwidth) of each set of links in multi-HMC systems and compares them with that of the single-HMC system.[10] From this figure, we observed that links between the two farthest HMCs (i.e., HMC_3 and HMC_4) serve only one fourth of the traffic serviced by the links between the host processor and the nearest HMC (i.e., HMC_1) on average. This indicates that HMC-to-HMC links have a higher opportunity to be turned off compared to host-to-HMC and HMC-to-host links. Our mechanism adapts to this behavior by disabling the former 19% more than the latter in LPM.

[10]Note that 30% of link bandwidth utilization does not necessarily mean that 70% of links can be disabled because off-chip memory accesses are often bursty (as shown in Figure 7.9) and CPU workloads are sensitive to memory latency as well as memory bandwidth.

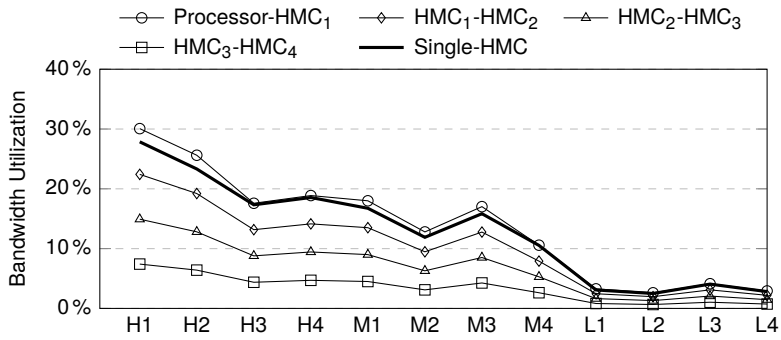


Figure 7.17: Link bandwidth utilization of four sets of links in multi-HMC systems (from 'Processor-HMC_1' to 'HMC_3-HMC_4') and one set of links in single-HMC systems ('Single-HMC'). Two-level prefetching is enabled in both cases.

We also found that, compared to the single-HMC system, the multi-HMC system shows slightly higher utilization of links between the processor and the nearest HMC. This is because, if a prefetch request initiates prefetches to multiple cache blocks that are spread across multiple HMCs, one packet for each target HMC needs to be generated in multi-HMC systems. However, its impact is limited to only 0.74% increase in link bandwidth utilization on average.

7.6 Summary

We proposed a power management scheme for off-chip links of hybrid memory cubes, which dynamically disables a subset of off-chip links for energy reduction. To determine the number of links to be disabled, we designed a low-cost hardware structure called link delay monitor, which estimates average link delays (as a proxy of performance degradation) under all possible link configurations at the same time. We also proposed two-level prefetching with in-HMC prefetch buffers to minimize unnecessary off-chip traffic, which in turn improves the effectiveness of our power management scheme. Experimental results show that our techniques reduce energy consumption of HMCs by 52% with performance degradation of 1.1% on average.

Chapter 8

Tesseract PIM System for Parallel Graph Processing

The recent emergence of *in-memory* big-data processing has initiated great interest in developing a hardware system that efficiently handles a large amount of data in main memory. There are two key challenges determining the performance of such systems: (1) how fast they can process each item and request the next item from memory, and (2) how fast the massive amount of data can be delivered from memory to computation units. Unfortunately, traditional computer architectures composed of heavy-weight cores and large on-chip caches are tailored for neither of these two challenges, thereby experiencing severe underutilization of existing hardware resources [131].

In order to tackle the first challenge, recent studies have proposed specialized on-chip accelerators for a limited set of operations, e.g., database systems [132–134], key-value stores [135], and stream processing [136]. Such accelerators mainly focus on improving core efficiency, thereby achieving better performance and energy efficiency compared to general-purpose cores, at the cost of generality. For example, Widx [133] is an on-chip accelerator for hash index lookups in main memory databases, which can be configured to accelerate either hash computation, index traversal, or output generation. Multiple Widx units can be used to exploit memory-level parallelism without the limitation of instruction window size, unlike conventional out-of-order processors.

This chapter is originally published in Proceedings of the International Symposium on Computer Architecture, 2015 [7].

Although specialized on-chip accelerators provide the benefit of computation efficiency, they impose a more fundamental challenge: *system performance does not scale well with the increase in the amount of data per server (or main memory capacity per server)*. This is because putting more accelerators provides speedup as long as the memory bandwidth is sufficient to feed them all. Unfortunately, memory bandwidth remains almost constant irrespective of memory capacity due to the pin count limitation per chip. For instance, Kocberber et al. [133] observe that using more than four index traversal units in Widx may not provide additional speedup due to off-chip bandwidth limitations. This implies that, in order to process twice the amount of data with the same performance, one needs to double the number of servers (which keeps memory bandwidth per unit data constant by limiting the amount of data in a server), rather than simply adding more memory modules to store data. Consequently, such approaches limit the memory capacity per server (or the amount of data handled by a single server) to achieve target performance, thereby leading to a relatively cost-ineffective and likely less scalable design as opposed to one that can enable increasing of memory bandwidth in a node along with more data in a node.

This scalability problem caused by the memory bandwidth bottleneck is expected to be greatly aggravated with the emergence of increasingly memory-intensive big-data workloads. One of the representative examples of this is large-scale graph analysis [2, 3, 137–141], which has recently been studied as an alternative to relational database based analysis for applications in, for example, social science, computational biology, and machine learning. Graph analysis workloads are known to put more pressure on memory bandwidth due to (1) large amounts of random memory accesses across large memory regions (leading to very limited cache efficiency) and (2) very small amounts of computation per item (leading to very limited ability to hide long memory latencies). These two characteristics make it very challenging to scale up such workloads despite their inherent parallelism, especially with conventional architectures based on large on-chip caches and scarce off-chip memory bandwidth.

In this chapter, we show that the processing-in-memory (PIM) can be a key enabler to realize *memory-capacity-proportional* performance in large-scale graph processing under the current pin count limitation. By putting computation units inside main memory, total memory bandwidth for the computation units scales well with the increase in memory capacity (and so does the computational power). Importantly, latency and energy overheads of moving data between computation units and main memory can be reduced as well. And, fortunately, such benefits can be realized in a cost-effective manner today through the 3D integration technology, which effectively combines logic

and memory dies, as opposed to the PIM architectures in 1990s, which suffered from the lack of an appropriate technology that could tightly couple logic and memory.

The key contributions of this chapter are as follows:

- We study an important domain of in-memory big-data processing workloads, large-scale graph processing, from the computer architecture perspective and show that memory bandwidth is the main bottleneck of such workloads.
- We provide the design and the programming interface of a new programmable accelerator for in-memory graph processing that can effectively utilize PIM using 3D-stacked memory technologies. Our new design is called Tesseract.[1]
- We develop an efficient mechanism for communication between different Tesseract cores based on message passing. This mechanism (1) enables effective hiding of long remote access latencies via the use of non-blocking message passing and (2) guarantees atomic memory updates without requiring software synchronization primitives.
- We introduce two new types of specialized hardware prefetchers that can fully utilize the available memory bandwidth with simple cores. These new designs take advantage of (1) the hints given by our new programming interface and (2) memory access characteristics of graph processing.
- We provide case studies of how five graph processing workloads can be mapped to our architecture and how they can benefit from it. Our evaluations show that Tesseract achieves 10x average performance improvement and 87% average reduction in energy consumption over a conventional high-performance baseline (a four-socket system with 32 out-of-order cores, having 640 GB/s of memory bandwidth), across five different graph processing workloads. Our evaluations use three large input graphs having four to seven million vertices, which are collected from real-world social networks and internet domains.

8.1 Background and Motivation

8.1.1 Large-Scale Graph Processing

A graph is a fundamental representation of relationship between objects. Examples of representative real-world graphs include social graphs, web graphs, transportation

[1]Tesseract means a four-dimensional hypercube. We named our architecture Tesseract because in-memory computation adds a new dimension to 3D-stacked memory technologies.

```
 1  for (v: graph.vertices) {
 2    v.pagerank = 1 / graph.num_vertices;
 3    v.next_pagerank = 0.15 / graph.num_vertices;
 4  }
 5  count = 0;
 6  do {
 7    diff = 0;
 8    for (v: graph.vertices) {
 9      value = 0.85 * v.pagerank / v.out_degree;
10      for (w: v.successors) {
11        w.next_pagerank += value;
12      }
13    }
14    for (v: graph.vertices) {
15      diff += abs(v.next_pagerank - v.pagerank);
16      v.pagerank = v.next_pagerank;
17      v.next_pagerank = 0.15 / graph.num_vertices;
18    }
19  } while (diff > e && ++count < max_iteration);
```

Figure 8.1: Pseudocode of PageRank computation.

graphs, and citation graphs. These graphs often have millions to billions of vertices with even larger numbers of edges, thereby making them difficult to be analyzed at high performance.

In order to tackle this problem, there exist several frameworks for large-scale graph processing by exploiting data parallelism [2,3,137–141]. Most of these frameworks focus on executing computation for different vertices in parallel while hiding synchronization from programmers to ease programmability. For example, the PageRank computation shown in Figure 8.1 can be accelerated by parallelizing the vertex loops [137] (lines 1–4, 8–13, and 14–18) since computation for each vertex is almost independent of each other. In this style of parallelization, synchronization is necessary to guarantee atomic updates of shared data (`w.next_pagerank` and `diff`) and no overlap between different vertex loops, which are automatically handled by the graph processing frameworks. Such an approach exhibits a high degree of parallelism, which is effective in processing graphs with billions of vertices.

Although graph processing algorithms can be parallelized through such frameworks, there are several issues that make efficient graph processing very challenging. First, graph processing incurs a large number of random memory accesses during neighbor traversal (e.g., line 11 of Figure 8.1). Second, graph algorithms show poor locality of

memory access since many of them access the entire set of vertices in a graph for each iteration. Third, memory access latency cannot be easily overlapped with computation because of the small amount of computation per vertex [2]. These aspects should be carefully considered when designing a system that can efficiently perform large-scale graph processing.

8.1.2 Graph Processing on Conventional Systems

Despite its importance, graph processing is a challenging task for conventional systems, especially when scaling to larger amounts of data (i.e., larger graphs). Figure 8.2 shows a scenario where one intends to improve graph processing performance of a server node equipped with out-of-order cores and DDR3-based main memory by adding more cores. We evaluate the performance of five workloads with 32 or 128 cores and with different memory interfaces (see Section 8.3 for our detailed evaluation methodology and the description of our systems). As the figure shows, simply increasing the number of cores is ineffective in improving performance significantly. Adopting a high-bandwidth alternative to DDR3-based main memory based on 3D-stacked DRAM, called Hybrid Memory Cube (HMC) [26], helps this situation to some extent, however, the speedups provided by using HMCs are far below the expected speedup from quadrupling the number of cores.

However, if we assume that cores can use the internal memory bandwidth of HMCs[2] ideally, i.e., without traversing the off-chip links, we can provide much higher performance by taking advantage of the larger number of cores. This is shown in the rightmost bars of Figure 8.3. The problem is that such high performance requires a massive amount of memory bandwidth (near 500 GB/s) as shown in Figure 8.2b. This is beyond the level of what conventional systems can provide under the current pin count limitations. What is worse, such a high amount of memory bandwidth is mainly consumed by random memory accesses over a large memory region, as explained in Section 8.1.1, which cannot be efficiently handled by the current memory hierarchies that are based on and optimized for data locality (i.e., large on-chip caches). This leads to the key question that we intend to answer in this work: *how can we provide such large amounts of memory bandwidth and utilize it for scalable and efficient graph processing in memory?*

[2]The term *internal memory bandwidth* indicates aggregate memory bandwidth provided by 3D-stacked DRAM. In our system composed of 16 HMCs, the internal memory bandwidth is 12.8 times higher than the off-chip memory bandwidth (see Section 8.3 for details).

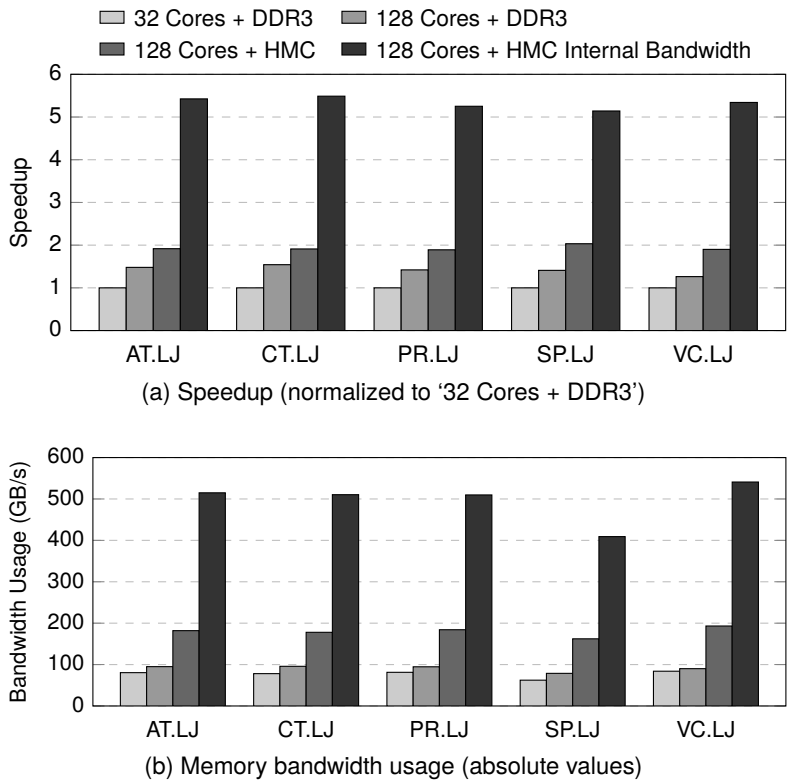


Figure 8.2: Performance of large-scale graph processing in conventional systems versus with ideal use of the HMC internal memory bandwidth.

8.1.3 Processing-in-Memory

To satisfy the high bandwidth requirement of large-scale graph processing workloads, we consider moving computation inside the memory, or *processing-in-memory*. The key objective of adopting PIM is not solely to provide high memory bandwidth, but especially to achieve *memory-capacity-proportional* bandwidth. Let us take the Hybrid Memory Cube [24] as a viable baseline platform for PIM. According to the HMC 1.0 specification [26], a single HMC provides up to 320 GB/s of *external* memory bandwidth through eight high-speed serial links. On the other hand, a 64-bit vertical interface for each DRAM partition (or *vault*, see Section 8.2.1 for details), 32 vaults per cube, and 2 Gb/s of TSV signaling rate [24] together achieve an *internal* memory bandwidth of 512 GB/s per cube. Moreover, this gap between external and internal memory bandwidth becomes much wider as the memory capacity increases with the use of more

HMCs. Considering a system composed of 16 8 GB HMCs as an example, conventional processors are still limited to 320 GB/s of memory bandwidth assuming that the CPU chip has the same number of off-chip links as that of an HMC. In contrast, PIM exposes 8 TB/s ($= 16 \times 512\,\mathrm{GB/s}$) of aggregate internal bandwidth to the in-memory computation units. This memory-capacity-proportional bandwidth facilitates scaling the system performance with increasing amount of data in a cost-effective way, which is a key concern in graph processing systems.

However, introducing a new processing paradigm brings a set of new challenges in designing a whole system. Throughout this chapter, we will answer three critical questions in designing a PIM system for graph processing: (1) How to design an architecture that can fully utilize internal memory bandwidth in an energy-efficient way, (2) how to communicate between different memory partitions (i.e., vaults) with a minimal impact on performance, and (3) how to design an expressive programming interface that reflects the hardware design.

8.2 Tesseract Architecture

8.2.1 Overview

Organization. Figure 8.3 shows a conceptual diagram of the proposed architecture. Although Tesseract does not rely on a particular memory organization, we choose the hybrid memory cube having eight 8 Gb DRAM layers (the largest device available in the current HMC specification [26]) as our baseline. An HMC, shown conceptually in Figure 8.3b is composed of 32 vertical slices (called *vaults*), eight 40 GB/s high-speed serial links as the off-chip interface, and a crossbar network that connects them. Each vault, shown in Figure 8.3c, is composed of a 16-bank DRAM partition and a dedicated memory controller.[3] In order to perform computation inside memory, a single-issue in-order core is placed at the logic die of each vault (32 cores per cube). In terms of area, a Tesseract core fits well into a vault due to the small size of an in-order core. For example, the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm^2 for each core [142]) corresponds to only 9.6% of the area of an 8 Gb DRAM die area (e.g., 226 mm^2 [143]).

Host-Tesseract Interface. In the proposed system, host processors have their own main memory (without PIM capability) and Tesseract acts like an accelerator that

[3]Due to the existence of built-in DRAM controllers, HMCs use a packet-based protocol for communication through the inter-/intra-HMC network instead of low-level DRAM commands as in DDRx protocols.

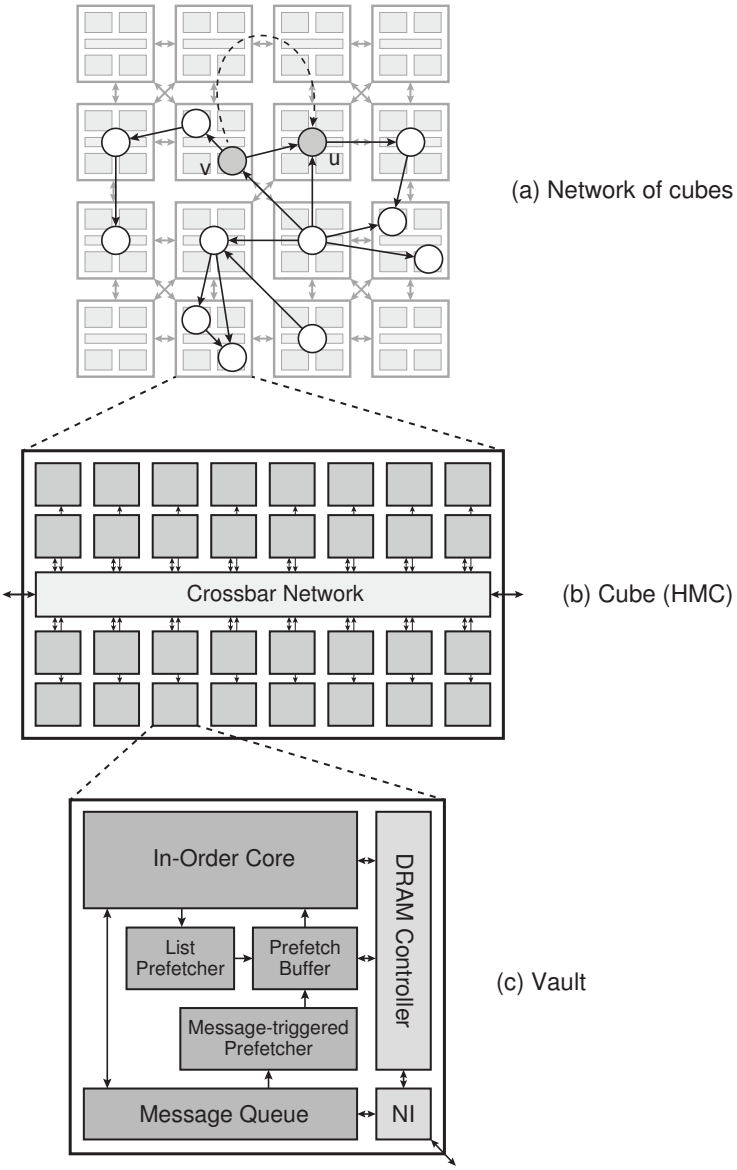


Figure 8.3: Tesseract architecture (the figure is not to scale).

is memory-mapped to part of a noncacheable memory region of the host processors. This eliminates the need for managing cache coherence between caches of the host processors and the 3D-stacked memory of Tesseract. Also, since in-memory big-data workloads usually do not require many features provided by virtual memory (along with the non-trivial performance overhead of supporting virtual memory) [144], Tesseract does not support virtual memory to avoid the need for address translation *inside* memory. Nevertheless, host processors can still use virtual addressing in their main memory since they use separate DRAM devices (apart from the DRAM of Tesseract) as their own main memory.[4]

Since host processors have access to the entire memory space of Tesseract, it is up to the host processors to distribute input graphs across HMC vaults. For this purpose, the host processors use a customized `malloc` call, which allocates an object (in this case, a vertex or a list of edges) to a specific vault. For example, `numa_alloc_onnode` in Linux (which allocates memory on a given NUMA node) can be extended to allocate memory on a designated vault. This information is exposed to applications since they use a single physical address space over all HMCs. An example of distributing an input graph to vaults is shown in Figure 8.3a. Algorithms to achieve a balanced distribution of vertices and edges to vaults are beyond the scope of this work. However, we analyze the impact of better graph distribution on the performance of Tesseract in Section 8.4.7.

Message Passing (Section 8.2.2). Unlike host processors that have access to the *entire* address space of the HMCs, each Tesseract core is restricted to access its own *local* DRAM partition only. Thus, a low-cost message passing mechanism is employed for communication between Tesseract cores. For example, vertex `v` in Figure 8.3a can remotely update a property of vertex `u` by sending a message that contains the target vertex id and the computation that will be done in the remote core (dotted line in Figure 8.3a). We choose message passing to communicate between Tesseract cores in order to: (1) avoid cache coherence issues among L1 data caches of Tesseract cores, (2) eliminate the need for locks to guarantee atomic updates of shared data, and (3) facilitate the hiding of remote access latencies through asynchronous message communication.

[4]For this purpose, Tesseract may adopt the direct segment approach [144] and interface its memory as a primary region. Supporting direct segment translation inside memory can be done simply by adding a small direct segment hardware for each Tesseract core and broadcasting the base, limit, and offset values from the host at the beginning of Tesseract execution.

Prefetching (Section 8.2.3). Although putting a core beneath memory exposes unprecedented memory bandwidth to the core, a single-issue in-order core design is far from the best way of utilizing this ample memory bandwidth. This is because such a core has to stall on each L1 cache miss. To enable better exploitation of the large amount of memory bandwidth while keeping the core simple, we design two types of simple hardware prefetchers: a list prefetcher and a message-triggered prefetcher. These are carefully tailored to the memory access patterns of graph processing workloads.

Programming Interface (Section 8.2.4). Importantly, we define a new programming interface that enables the use of our system. Our programming interface is easy to use, yet general enough to express many different graph algorithms.

8.2.2 Remote Function Call via Message Passing

Tesseract moves computation to the target core that contains the data to be processed, instead of allowing remote memory accesses. For simplicity and generality, we implement computation movement as a remote function call [145, 146]. In this section, we propose two different message passing mechanisms, both of which are supported by Tesseract: blocking remote function call and non-blocking remote function call.

Blocking Remote Function Call. A blocking remote function call is the most intuitive way of accessing remote data. In this mechanism, a local core requests a remote core to (1) execute a specific function remotely and (2) send the return value back to the local core. The exact sequence of performing a blocking remote function call is as follows:

1. The local core sends a packet containing the function address[5] and function arguments[6] to the remote core and waits for its response.

2. Once the packet arrives at the remote vault, the network interface stores function arguments to the special registers visible from the core and emits an interrupt for the core.

3. The remote core executes the function in *interrupt mode*, writes the return value to a special register, and switches back to the normal execution mode.

4. The remote core sends the return value back to the local core.

[5]We assume that all Tesseract cores store the same code into the same location of their local memory so that function addresses are compatible across different Tesseract cores.

[6]In this work, we restrict the maximum size of arguments to be 32 bytes, which should be sufficient for general use. We also provide an API to transfer data larger than 32 bytes in Section 8.2.4.

Note that the execution of a remote function call is *not* preempted by another remote function call in order to guarantee atomicity. Also, cores may temporarily disable interrupt execution to modify data that might be accessed by blocking remote function calls.

This style of remote data access is useful for global state checks. For example, checking the condition '`diff > e`' in line 19 of Figure 8.1 can be done using this mechanism. However, it may not be the performance-optimal way of accessing remote data because (1) local cores are blocked until responses arrive from remote cores and (2) each remote function call emits an interrupt, incurring the latency overhead of context switching. This motivates the need for another mechanism for remote data access, a *non-blocking* remote function call.

Non-Blocking Remote Function Call. A non-blocking remote function call is semantically similar to its blocking counterpart, except that it cannot have return values. This simple restriction greatly helps to optimize the performance of remote function calls in two ways.

First, a local core can continue its execution after invoking a non-blocking remote function call since the core does not have to wait for the termination of the function. In other words, it allows hiding remote access latency because sender cores can perform their own work while messages are being transferred and processed. However, this makes it impossible to figure out whether or not the remote function call is finished. To simplify this problem, we ensure that all non-blocking remote function calls do not cross synchronization barriers. In other words, results of remote function calls are guaranteed to be visible after the execution of a barrier. Similar consistency models can be found in other parallelization frameworks such as OpenMP [147].

Second, since the execution of non-blocking remote function calls can be delayed, batch execution of such functions is possible by buffering them and executing all of them with a single interrupt. For this purpose, we add a *message queue* to each vault that stores messages for non-blocking remote function calls. Functions in this queue are executed once either the queue is full or a barrier is reached. Batching the execution of remote function calls helps to avoid the latency overhead of context switching incurred by frequent interrupts.

Non-blocking remote function calls are mainly used for updating remote data. For example, updating PageRank values of remote vertices in line 11 of Figure 8.1 can be implemented using this mechanism. Note that, unlike the original implementation where locks are required to guarantee atomic updates of `w.next_pagerank`, our mechanism eliminates the need for locks or other synchronization primitives since it guarantees that

(1) only the local core of vertex `w` can access and modify its property and (2) remote function call execution is not preempted by other remote function calls.

8.2.3 Prefetching

We develop two prefetching mechanisms to enable each Tesseract core to exploit the high available memory bandwidth.

List Prefetching. One of the most common memory access patterns is sequential accesses with a constant stride. Such access patterns are found in graph processing as well. For example, most graph algorithms frequently traverse the list of vertices and the list of edges for each vertex (e.g., the **`for`** loops in Figure 8.1), resulting in strided access patterns.

Memory access latency of such a simple access pattern can be easily hidden by employing a stride prefetcher. In this work, we use a stride prefetcher based on a reference prediction table (RPT) [148] that prefetches multiple cache blocks ahead to utilize the high memory bandwidth. In addition, we modify the prefetcher to accept information about the start address, the size, and the stride of each list from the application software. Such information is recorded in the four-entry list table at the beginning of a loop and is removed from it at the end of the loop. Inside the loop, the prefetcher keeps track of only the memory regions registered in the list table and installs an RPT entry if the observed stride conforms to the hint. An RPT entry is removed once it reaches the end of the memory region.

Message-triggered Prefetching. Although stride prefetchers can cover frequent sequential accesses, graph processing often involves a large amount of random access patterns. This is because, in graph processing, information flows through the edges, which requires pointer chasing over edges toward randomly-located target vertices. Such memory access patterns cannot be easily predicted by stride prefetchers.

Interestingly, most of the random memory accesses in graph processing happen on remote accesses (i.e., neighbor traversal). This motivates the second type of prefetching we devise, called *message-triggered prefetching*, shown in Figure 8.4. The key idea is to prefetch data that will be accessed during a non-blocking remote function call *before* the execution of the function call. For this purpose, we add an optional field for each non-blocking remote function call packet, indicating a memory address to be prefetched. As soon as a request containing the prefetch hint is inserted into the message queue, the message-triggered prefetcher issues a prefetch request based on the hint and marks the message as ready when the prefetch is serviced. When more than a predetermined

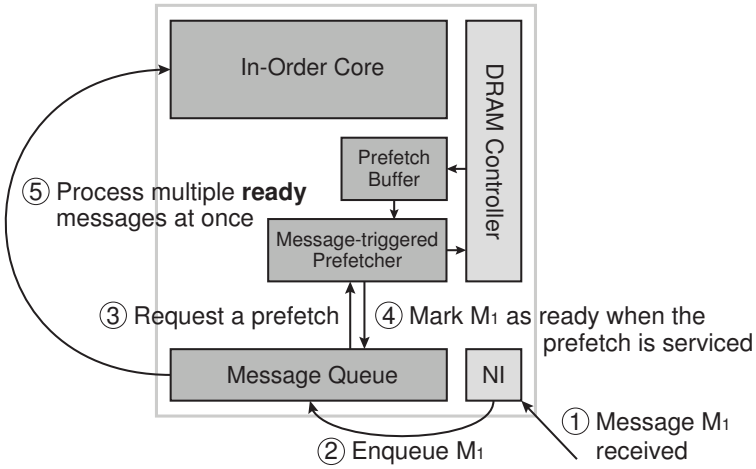


Figure 8.4: Message-triggered prefetching mechanism.

number (M_{th}) of messages in the message queue are ready, the message queue issues an interrupt to the core to process the *ready* messages.[7]

Message-triggered prefetching is unique in two aspects. First, it can eliminate processor stalls due to memory accesses inside remote function call execution by processing only ready messages. This is achieved by exploiting the time slack between the arrival of a non-blocking remote function call message and the time when the core starts servicing the message. Second, it can be *exact*, unlike many other prefetching techniques, since graph algorithms use non-blocking remote function calls to send updates over edges, which contain the *exact* memory addresses of the target vertices. For example, a non-blocking remote function call for line 11 of Figure 8.1 can provide the address of `w.next_pagerank` as a prefetch hint, which is exact information on the address instead of a prediction that can be incorrect.

Prefetch Buffer. The two prefetch mechanisms store prefetched blocks into prefetch buffers [111] instead of L1 caches. This is to prevent the situation where prefetched blocks are evicted from the L1 cache before they are referenced due to the long interval between prefetch requests and their demand accesses. For instance, a cache block loaded by message-triggered prefetching has to wait to be accessed until at least M_{th} messages are ready. Meanwhile, other loads inside the normal execution mode may evict the block according to the replacement policy of the L1 cache. A similar effect can be observed

[7] If the message queue becomes full or a barrier is reached before M_{th} messages are ready, *all* messages are processed regardless of their readiness.

when loop execution with list prefetching is preempted by a series of remote function call executions.

8.2.4 Programming Interface

In order to utilize the new Tesseract design, we provide the following primitives for programming in Tesseract. We introduce several major API calls for Tesseract: `get`, `put`, `disable_interrupt`, `enable_interrupt`, `copy`, `list_begin`, `list_end`, and `barrier`. Hereafter, we use **`A`** and **`S`** to indicate the memory address type (e.g., `void*` in C) and the size type (e.g., `size_t` in C), respectively.

```
get(id, A func, A arg, S arg_size, A ret, S ret_size)
put(id, A func, A arg, S arg_size, A prefetch_addr)
```

`get` and `put` calls represent blocking and non-blocking remote function calls, respectively. The id of the target remote core is specified by the `id` argument.[8] The start address and the size of the function argument is given by `arg` and `arg_size`, respectively, and the return value (in the case of `get`) is written to the address `ret`. In the case of `put`, an optional argument `prefetch_addr` can be used to specify the address to be prefetched by the message-triggered prefetcher.

```
disable_interrupt()
enable_interrupt()
```

`disable_interrupt` and `enable_interrupt` calls guarantee that the execution of instructions enclosed by them are not preempted by interrupts from remote function calls. This prevents data races between normal execution mode and interrupt mode as explained in Section 8.2.2.

```
copy(id, A local, A remote, S size)
```

The `copy` call implements copying a local memory region to a remote memory region. It is used instead of `get` or `put` commands if the size of transfer exceeds the maximum size of arguments. This command is guaranteed to take effect before the nearest barrier synchronization (similar to the `put` call).

```
list_begin(A address, S size, S stride)
list_end(A address, S size, S stride)
```

[8] If a core issues a `put` command with its own id, it can either be replaced by a simple function call or use the same message queue mechanism as in remote messages. In this work, we insert local messages to the message queue only if message-triggered prefetching (Section 8.2.3) is available so that the prefetching can be applied to local messages as well.

```
1  ...
2  count = 0;
3  do {
4    ...
5    list_for (v: graph.vertices) {
6      value = 0.85 * v.pagerank / v.out_degree;
7      list_for (w: v.successors) {
8        arg = (w, value);
9        put(w.id, function(w, value) {
10         w.next_pagerank += value;
11       }, &arg, sizeof(arg), &w.next_pagerank);
12     }
13   }
14   barrier();
15   ...
16 } while (diff > e && ++count < max_iteration);
```

Figure 8.5: PageRank computation in Tesseract.

`list_begin` and `list_end` calls are used to update the list table, which contains hints for list prefetching. Programmers can specify the start address of a list, the size of the list, and the size of an item in the list (i.e., stride) to initiate list prefetching.

```
barrier()
```

The `barrier` call implements a synchronization barrier across all Tesseract cores. One of the cores in the system (predetermined by designers or by the system software) works as a master core to collect the synchronization status of each core.

8.2.5 Application Mapping

Figure 8.5 shows the PageRank computation using our programming interface (recall that the original version was shown in Figure 8.1). We only show the transformation for lines 8–13 of Figure 8.1, which contain the main computation. `list_for` is used as an abbreviation of a **`for`** loop surrounded by `list_begin` and `list_end` calls.

Most notably, remote memory accesses for updating the `next_pagerank` field are transformed into `put` calls. Consequently, unlike the original implementation where every L1 cache miss or lock contention for `w.next_pagerank` stalls the core, our implementation facilitates cores to (1) continuously issue `put` commands without being blocked by cache misses or lock acquisition and (2) promptly update PageRank values without stalls due to L1 cache misses through message-triggered prefetching. List

prefetching also helps to achieve the former objective by prefetching pointers to the successor vertices (i.e., the list of outgoing edges).

We believe that such transformation is simple enough to be easily integrated into existing graph processing frameworks [2, 3, 138–141] or DSL compilers for graph processing [137, 149]. This is a part of our future work.

8.3 Evaluation Methodology

8.3.1 Simulation Configuration

We evaluate our architecture using an in-house cycle-accurate x86-64 simulator whose frontend is Pin [95]. The simulator has a cycle-level model of many microarchitectural components, including in-order/out-of-order cores considering register/structural dependencies, multi-bank caches with limited numbers of MSHRs, MESI cache coherence, DDR3 controllers, and HMC links. Our simulator runs multithreaded applications by inspecting pthread APIs for threads and synchronization primitives. For Tesseract, it also models remote function calls by intercepting get/put commands (manually inserted into software) and injecting messages into the timing model accordingly. The rest of this subsection briefly describes the system configuration used for our evaluations.

DDR3-Based System. We model a high-performance conventional DDR3-based system with 32 4 GHz four-wide out-of-order cores, each with a 128-entry instruction window and a 64-entry load-store queue (denoted as *DDR3-OoO*). Each socket contains eight cores and all four sockets are fully connected with each other by high-speed serial links, providing 40 GB/s of bandwidth per link. Each core has 32 KB L1 instruction/data caches and a 256 KB L2 cache, and eight cores in a socket share an 8 MB L3 cache. All three levels of caches are non-blocking, having 16 (L1), 16 (L2), and 64 (L3) MSHRs [150]. Each L3 cache is equipped with a feedback-directed prefetcher with 32 streams [112]. The main memory has 128 GB of memory capacity and is organized as two channels per CPU socket, four ranks per channel, eight banks per rank, and 8 KB rows with timing parameters of DDR3-1600 11-11-11 devices [96], yielding 102.4 GB/s of memory bandwidth exploitable by cores.

DDR3-OoO resembles modern commodity servers composed of multi-socket, high-end CPUs backed by DDR3 main memory. Thus, we choose it as the baseline of our evaluations.

HMC-Based System. We use two different types of cores for the HMC-based system: *HMC-OoO*, which consists of the same cores used in DDR3-OoO, and *HMC-MC*, which

is comprised of 512 2 GHz single-issue in-order cores (128 cores per socket), each with 32 KB L1 instruction/data caches and no L2 cache. For the main memory, we use 16 8 GB HMCs (128 GB in total, 32 vaults per cube, 16 banks per vault [26], and 256 B pages) connected with the processor-centric topology proposed by Kim et al. [116]. The total memory bandwidth exploitable by the cores is 640 GB/s.

HMC-OoO and HMC-MC represent future server designs based on emerging memory technologies. They come with two flavors, one with few high-performance cores and the other with many low-power cores, in order to reflect recent trends in commercial server design.

Tesseract System. Our evaluated version of the Tesseract paradigm consists of 512 2 GHz single-issue in-order cores, each with 32 KB L1 instruction/data caches and a 32-entry message queue (1.5 KB), one for each vault of the HMCs. We conservatively assume that entering or exiting the interrupt mode takes 50 processor cycles (or 25 ns). We use the same number of HMCs (128 GB of main memory capacity) as that of the HMC-based system and connect the HMCs with the Dragonfly topology as suggested by previous work [116]. Each vault provides 16 GB/s of internal memory bandwidth to the Tesseract core, thereby reaching 8 TB/s of total memory bandwidth exploitable by Tesseract cores. We do not model the host processors as computation is done entirely inside HMCs without intervention from host processors.

For our prefetching schemes, we use a 4 KB 16-way set-associative prefetch buffer for each vault. The message-triggered prefetcher handles up to 16 prefetches and triggers the message queue to start processing messages when more than 16 ($= M_{\text{th}}$) messages are ready. The list prefetcher is composed of a four-entry list table and a 16-entry reference prediction table (0.48 KB) and is set to prefetch up to 16 cache blocks ahead. M_{th} and the prefetch distance of the list prefetcher are determined based on our experiments on a limited set of configurations. Note that comparison of our schemes against other software prefetching approaches is hard to achieve because Tesseract is a message-passing architecture (i.e., each core can access its local DRAM partition only), and thus, existing mechanisms require significant modifications to be applied to Tesseract to prefetch data stored in remote memory.

8.3.2 Workloads

We implemented five graph algorithms in C++. Average Teenager Follower (AT) computes the average number of teenage followers of users over k years old [149]. Conductance (CT) counts the number of edges crossing a given partition X and its

complement X^c [137,149]. PageRank (PR) is an algorithm that evaluates the importance of web pages [2, 137, 149, 151]. Single-Source Shortest Path (SP) finds the shortest path from the given source to each vertex [2, 149]. Vertex Cover (VC) is an approximation algorithm for the minimum vertex cover problem [137]. Due to the long simulation times, we simulate only one iteration of PR, four iterations of SP, and one iteration of VC. Other algorithms are simulated to the end.

Since runtime characteristics of graph processing algorithms could depend on the shapes of input graphs, we use three real-world graphs as inputs of each algorithm: *ljournal-2008* from the LiveJournal social site (LJ, $|V| = 5.3$ M, $|E| = 79$ M), *enwiki-2013* from the English Wikipedia (WK, $|V| = 4.2$ M, $|E| = 101$ M), and *indochina-2004* from the country domains of Indochina (IC, $|V| = 7.4$ M, $|E| = 194$ M) [152]. These inputs yield 3–5 GB of memory footprint, which is much larger than the total cache capacity of any system in our evaluations. Although larger datasets cannot be used due to the long simulation times, our evaluation with relatively smaller memory footprints is conservative as it penalizes Tesseract because conventional systems in our evaluations have much larger caches (41 MB in HMC-OoO) than the Tesseract system (16 MB). The input graphs used in this work are known to share similar characteristics with large real-world graphs in terms of their small diameters and power-law degree distributions [153].[9]

8.4 Evaluation Results

8.4.1 Performance

Figure 8.6 compares the performance of the proposed Tesseract system against that of conventional systems (DDR3-OoO, HMC-OoO, and HMC-MC). In this figure, LP and MTP indicate the use of list prefetching and message-triggered prefetching, respectively. The last set of bars, labeled as GM, indicates geometric mean across all workloads.

Our evaluation results show that Tesseract outperforms the DDR3-based conventional architecture (DDR3-OoO) by 9x even without prefetching techniques. Replacing the DDR3-based main memory with HMCs (HMC-OoO) and using many in-order cores instead of out-of-order cores (HMC-MC) bring only marginal performance improvements over the conventional systems.

[9] We conducted a limited set of experiments with even larger graphs (*it-2004*, *arabic-2005*, and *uk-2002* [152], $|V| = 41$ M/23 M/19 M, $|E| = 1151$ M/640 M/298 M, 32 GB/18 GB/10 GB of memory footprints, respectively) and observed similar trends in performance and energy efficiency.

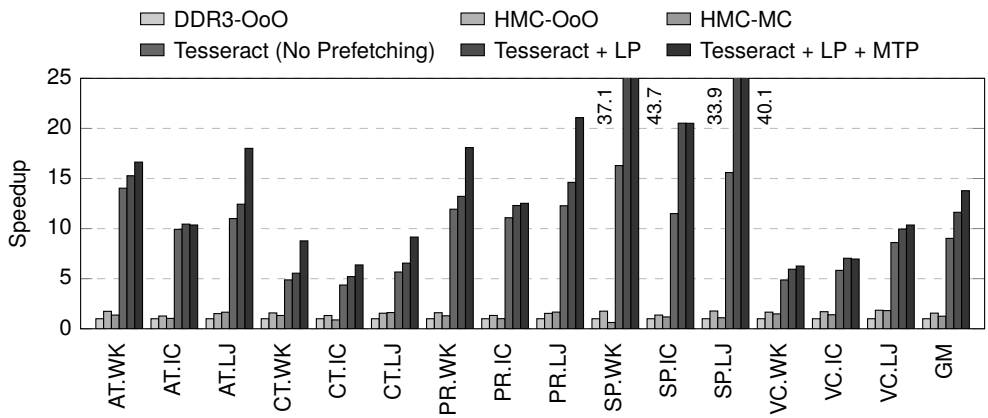


Figure 8.6: Performance comparison between conventional architectures and Tesseract (normalized to DDR3-OoO).

Our prefetching mechanisms, when employed together, enable Tesseract to achieve a 14x average performance improvement over the DDR3-based conventional system, while minimizing the storage overhead to less than 5 KB per core (see Section 8.3.1). Message-triggered prefetching is particularly effective in graph algorithms with large numbers of neighbor accesses (e.g., CT, PR, and SP), which are difficult to handle efficiently in conventional architectures.

The reason why conventional systems fall behind Tesseract is that they are limited by the low off-chip link bandwidth (102.4 GB/s in DDR3-OoO or 640 GB/s in HMC-OoO/-MC) whereas our system utilizes the large internal memory bandwidth of HMCs (8 TB/s).[10] Perhaps more importantly, such bandwidth discrepancy becomes even more pronounced as the main memory capacity per server gets larger. For example, doubling the memory capacity linearly increases the memory bandwidth in our system, while the memory bandwidth of the conventional systems remains the same.

To provide more insight into the performance improvement of Tesseract, Figure 8.7 shows memory bandwidth usage and average memory access latency of each system (we omit results for workloads with WK and IC datasets for brevity). As the figure shows, the amount of memory bandwidth utilized by Tesseract is in the order of several TB/s,

[10]Although Tesseract also uses off-chip links for remote accesses, moving computation to where data reside (i.e., using the remote function calls in Tesseract) consumes much less bandwidth than fetching data to computation units. For example, the minimum memory access granularity of conventional systems is one cache block (typically 64 bytes), whereas each message in Tesseract consists of a function pointer and small-sized arguments (up to 32 bytes). Sections 8.4.5 and 8.4.6 discuss the impact of off-chip link bandwidth on Tesseract performance.

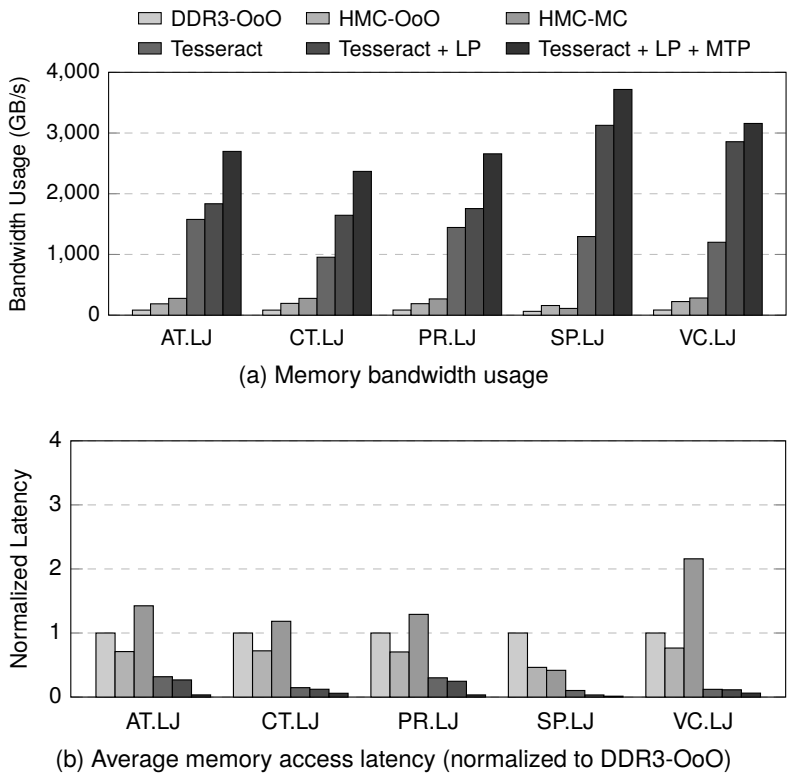


Figure 8.7: Memory characteristics of graph processing workloads in conventional architectures and Tesseract.

which is clearly beyond the level of what conventional architectures can reach even with advanced memory technologies. This, in turn, greatly affects the average memory access latency, leading to a 96% lower memory access latency in our architecture compared to the DDR3-based system. This explains the main source of the large speedup achieved by our system.

Figure 8.7a also provides support for our decision to have one-to-one mapping between cores and vaults. Since the total memory bandwidth usage does not reach its limit (8 TB/s), allocating multiple vaults to a single core could cause further imbalance between computation power and memory bandwidth. Also, putting more than one core per vault complicates the system design in terms of higher thermal density, degraded quality of service due to sharing of one memory controller between multiple cores, and potentially more sensitivity to placement of data. For these reasons, we choose to employ one core per vault.

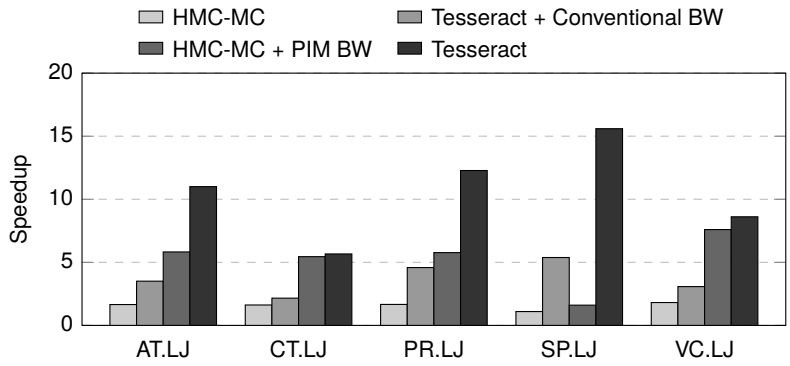


Figure 8.8: HMC-MC and Tesseract under the same bandwidth.

8.4.2 Iso-Bandwidth Comparison

In order to dissect the performance impact of increased memory bandwidth and our architecture design, we perform idealized limit studies of two new configurations: (1) HMC-MC utilizing the internal memory bandwidth of HMCs *without* off-chip bandwidth limitations (called *HMC-MC + PIM BW*) and (2) Tesseract, implemented on the host side, leading to severely constrained by off-chip link bandwidth (called *Tesseract + Conventional BW*). The first configuration shows the ideal performance of conventional architectures without any limitation due to off-chip bandwidth. The second configuration shows the performance of Tesseract if it were limited by conventional off-chip bandwidth. Note that HMC-MC has the same core and cache configuration as that of Tesseract. For fair comparison, prefetching mechanisms of Tesseract are disabled. We also show the performance of regular HMC-MC and Tesseract (the leftmost and the rightmost bars in Figure 8.8).

As shown in Figure 8.8, simply increasing the memory bandwidth of conventional architectures is not sufficient for them to reach the performance of Tesseract. Even if the memory bandwidth of HMC-MC is artificially provisioned to the level of Tesseract, Tesseract still outperforms HMC-MC by 2.2x even without prefetching. Considering that HMC-MC has the same number of cores and the same cache capacity as those of Tesseract, we found that this improvement comes from our programming model that can overlap long memory access latency with computation through non-blocking remote function calls. The performance benefit of our new programming model is also confirmed when we compare the performance of *Tesseract + Conventional BW* with that of HMC-MC. We observed that, under the conventional bandwidth limitation, Tesseract provides 2.3x the performance of HMC-MC, which is 2.8x less speedup compared to

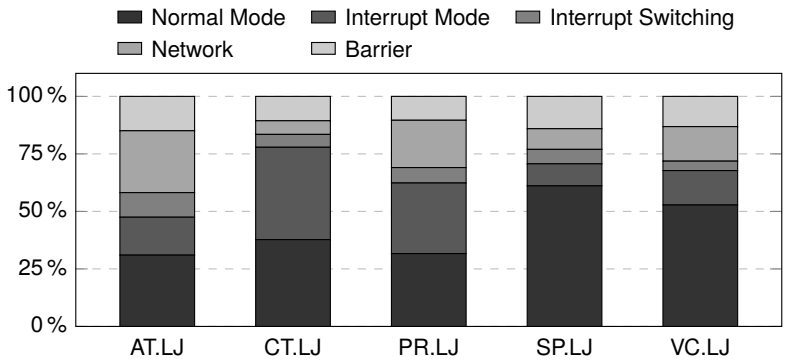


Figure 8.9: Execution time breakdown of our architecture.

its PIM version. This implies that the use of PIM and our new programming model are roughly of equal importance in achieving the performance of Tesseract.

8.4.3 Execution Time Breakdown

Figure 8.9 shows execution time broken down into each operation in Tesseract (with prefetching mechanisms), averaged over all cores in the system. In many workloads, execution in normal execution mode and interrupt mode dominates the total execution time. However, in some applications, up to 26% of execution time is spent on waiting for network due to a significant amount of off-chip communication caused by neighbor traversal. Since neighbor traversal uses *non-blocking* remote function calls, the time spent is essentially due to the network backpressure incurred as a result of limited network bandwidth. In Sections 8.4.6 and 8.4.7, we show how this problem can be mitigated by either increased off-chip bandwidth or better graph distribution schemes.

In addition, some workloads spend notable execution time waiting for barrier synchronization. This is due to workload imbalance across cores in the system. This problem can be alleviated by employing better data mapping (e.g., graph partitioning based vertex distribution, etc.), which is orthogonal to our proposed system.

8.4.4 Prefetch Efficiency

Figure 8.10 shows two metrics to evaluate the efficiency of our prefetching mechanisms. First, to evaluate prefetch timeliness, it compares our scheme against an ideal one where all prefetches are serviced instantly (in zero cycles) without incurring DRAM contention. Second, it depicts prefetch coverage, i.e., ratio of prefetch buffer hits over all L1 cache misses.

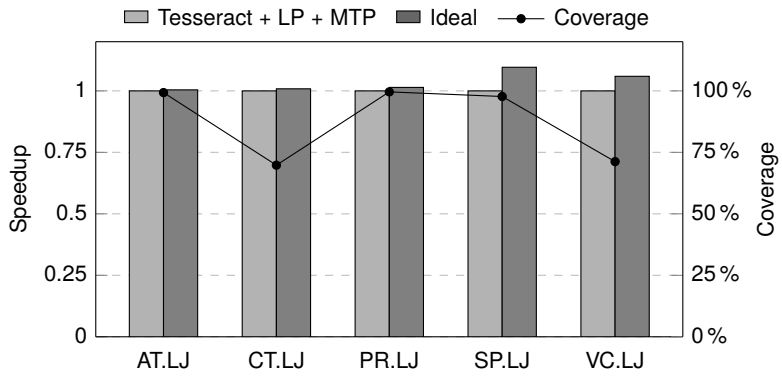


Figure 8.10: Efficiency of our prefetching mechanisms.

We observed that our prefetching schemes perform within 1.8% of their ideal implementation with perfect timeliness and no bandwidth contention. This is very promising, especially considering that pointer chasing in graph processing is not a prefetch-friendly access pattern. The reason why our message-triggered prefetching shows such good timeliness is that it utilizes the slack between message arrival time and message processing time. Thus, as long as there is enough slack, our proposed schemes can fully hide the DRAM access latency. Our experimental results indicate that, on average, each message stays in the message queue for 1400 Tesseract core cycles (i.e., 700 ns) before they get processed (not shown), which is much longer than the DRAM access latency in most cases.

Figure 8.10 also shows that our prefetching schemes cover 87% of L1 cache misses, on average. The coverage is high because our prefetchers tackle two major sources of memory accesses in graph processing, namely vertex/edge list traversal and neighbor traversal, with *exact* information from domain-specific knowledge provided as software hints.

We conclude that the new prefetching mechanisms can be very effective in our Tesseract design for graph processing workloads.

8.4.5 Scalability

Figure 8.11 evaluates the scalability of Tesseract by measuring the performance of 32/128/512-core systems (i.e., systems with 8/32/128 GB of main memory in total), normalized to the performance of the 32-core Tesseract system. Tesseract provides nearly ideal scaling of performance when the main memory capacity is increased from 8 GB to 32 GB. On the contrary, further quadrupling the main memory capacity to

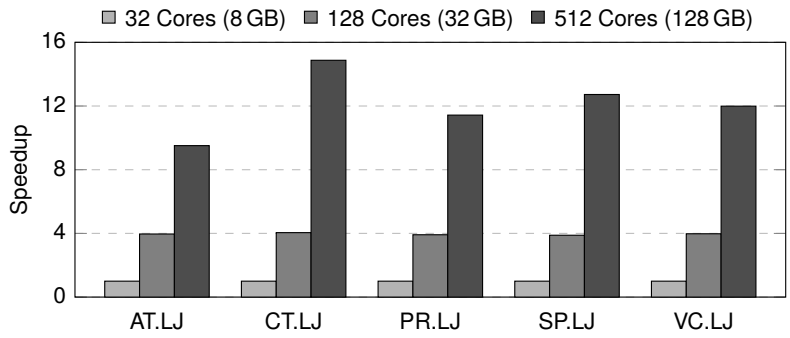


Figure 8.11: Performance scalability of Tesseract.

128 GB shows less optimal performance compared to ideal scaling. The cause of this is that, as more cubes are added into our architecture, off-chip communication overhead becomes more dominant due to remote function calls. For example, as the number of Tesseract cores increases from 128 to 512, the average bandwidth consumption of the busiest off-chip link in Tesseract increases from 8.5 GB/s to 17.2 GB/s (i.e., bandwidth utilization of the busiest link increases from 43% to 86%) in the case of AT.LJ. However, it should be noticed that, despite this sublinear performance scaling, increasing the main memory capacity widens the performance gap between conventional architectures and ours even beyond 128 GB since conventional architectures do not scale well with the increasing memory capacity. We believe that optimizing the off-chip network and data mapping will further improve scalability of our architecture. We discuss these in the next two sections.

8.4.6 Effect of Higher Off-Chip Network Bandwidth

The recent HMC 2.0 specification boosts the off-chip memory bandwidth from 320 GB/s to 480 GB/s [28]. In order to evaluate the impact of such an increased off-chip bandwidth on both conventional systems and Tesseract, we evaluate HMC-OoO and Tesseract with 50% higher off-chip bandwidth. As shown in Figure 8.12, such improvement in off-chip bandwidth widens the gap between HMC-OoO and Tesseract in graph processing workloads which intensively use the off-chip network for neighbor traversal. This is because the 1.5x off-chip link bandwidth is still *far below* the memory bandwidth required by large-scale graph processing workloads in conventional architectures (see Figure 8.7a). However, 1.5x off-chip bandwidth greatly helps to reduce network-induced stalls in Tesseract, enabling even more efficient utilization of internal memory bandwidth. We observed that, with this increase in off-chip link bandwidth, graph processing in

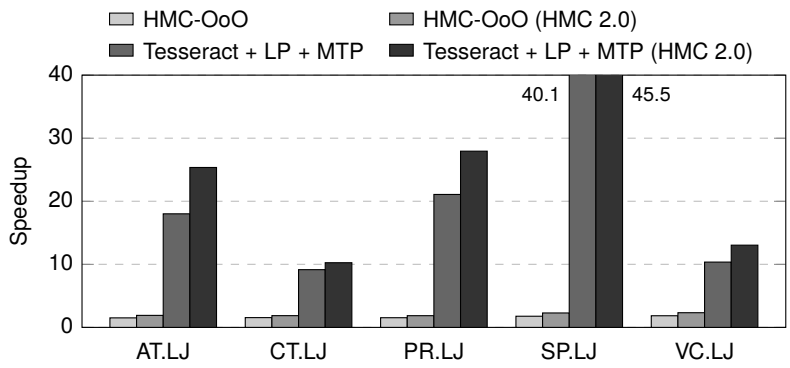


Figure 8.12: System performance under HMC 2.0 specification.

Tesseract scales better to 512 cores (not shown: 14.9x speedup resulting from 16x more cores, going from 32 to 512 cores).

8.4.7 Effect of Better Graph Distribution

Another way to improve off-chip transfer efficiency is to employ better data partitioning schemes that can minimize communication between different vaults. In order to analyze the effect of data partitioning on system performance, Figure 8.13 shows the performance improvement of Tesseract when the input graphs are distributed across vaults based on graph partitioning algorithms. For this purpose, we use METIS [154] to perform 512-way multi-constraint partitioning to balance the number of vertices, outgoing edges, and incoming edges of each partition, as done in a recent previous work [139]. The evaluation results do not include the execution time of the partitioning algorithm to clearly show the impact of graph distribution on graph analysis performance.

Employing better graph distribution can further improve the performance of Tesseract. This is because graph partitioning minimizes the number of edges crossing between different partitions (53% fewer edge cuts compared to random partitioning in LJ), and thus, reduces off-chip network traffic for remote function calls. For example, in AT.LJ, the partitioning scheme eliminates 53% of non-blocking remote function calls compared to random partitioning (which is our baseline).

However, in some workloads, graph partitioning shows only small performance improvement (CT.LJ) or even degrades performance (SP.LJ) over random partitioning. This is because graph partitioning algorithms are unaware of the amount of work per vertex, especially when it changes over time. As a result, they can exacerbate the workload imbalance across vaults. A representative example of this is the shortest

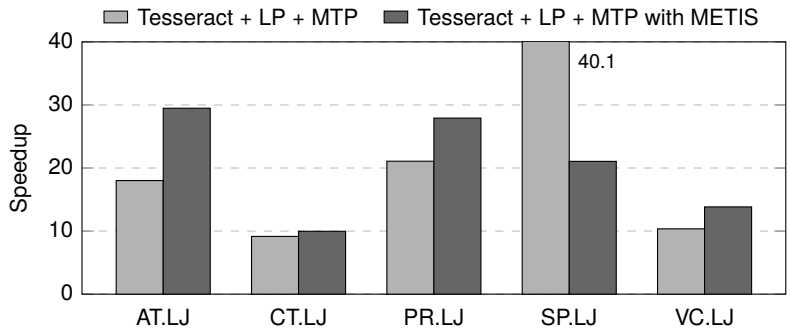


Figure 8.13: Performance improvement after graph partitioning.

path algorithm (SP.LJ), which skips computation for vertices whose distances did not change during the last iteration. This algorithm experiences severe imbalance at the beginning of execution, where vertex updates happen mostly within a single partition. This is confirmed by the observation that Tesseract with METIS spends 59% of execution time waiting for synchronization barriers. This problem can be alleviated with migration-based schemes, which will be explored in our future work.

8.4.8 Energy/Power Consumption and Thermal Analysis

Figure 8.14 shows the normalized energy consumption of HMCs in HMC-based systems including Tesseract. We model the power consumption of logic/memory layers and Tesseract cores by leveraging previous work [27], which is based on Micron's disclosure, and scaling the numbers as appropriate for our configuration. Tesseract consumes 87% less average energy compared to conventional HMC-based systems with out-of-order cores, mainly due to its shorter execution time. The dominant portion of the total energy consumption is from the SerDes circuits for off-chip links in both HMC-based systems and Tesseract (62% and 45%, respectively), while Tesseract cores contribute 15% of the total energy consumption.

Tesseract increases the average power consumption (not shown) by 40% compared to HMC-OoO mainly due to the in-order cores inside it and the higher DRAM utilization. Although the increased power consumption may have a negative impact on device temperature, the power consumption is expected to be still within the power budget according to a recent industrial research on thermal feasibility of 3D-stacked PIM [59]. Specifically, assuming that a logic die of the HMC has the same area as an 8 Gb DRAM die (e.g., 226 mm^2 [143]), the highest power density of the logic die across all workloads in our experiments is 94 mW/mm^2 in Tesseract, which remains below the maximum

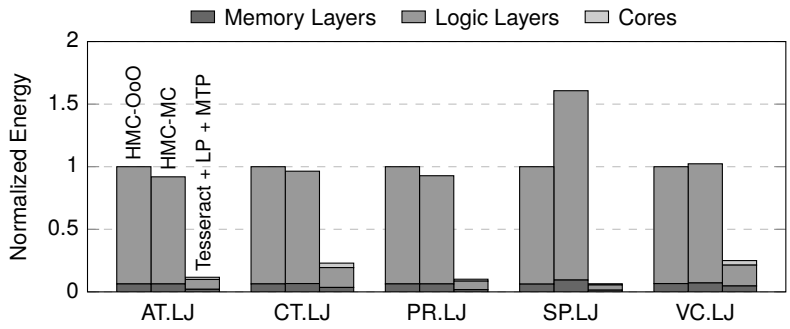


Figure 8.14: Normalized energy consumption of HMCs.

power density that does not require faster DRAM refresh using a passive heat sink (i.e., 133 mW/mm^2 [59]).

We conclude that Tesseract is thermally feasible and leads to greatly reduced energy consumption on state-of-the-art graph processing workloads.

8.5 Summary

In this chapter, we revisited the processing-in-memory concept in the completely new context of (1) cost-effective integration of logic and memory through 3D stacking and (2) emerging large-scale graph processing workloads that require an unprecedented amount of memory bandwidth. To this end, we introduced a programmable PIM accelerator for large-scale graph processing, called Tesseract. Our new system features (1) many in-order cores inside a memory chip, (2) a new message passing mechanism that can hide remote access latency within our PIM design, (3) new hardware prefetchers specialized for graph processing, and (4) a programming interface that exploits our new hardware design. We showed that Tesseract greatly outperforms conventional high-performance systems in terms of both performance and energy efficiency. Perhaps more importantly, Tesseract achieves *memory-capacity-proportional* performance, which is the key to handling increasing amounts of data in a cost-effective manner. We conclude that our new design can be an efficient and scalable substrate to execute emerging data-intensive applications with intense memory bandwidth demands.

Chapter 9

PIM-Enabled Instructions

To date, most of the existing PIM architectures are based on general-purpose computation units inside memory for flexibility across different applications [16–18, 20–22, 27, 58, 68, 156–158]. For example, Tesseract integrates specialized in-order cores into main memory in order to facilitate various graph algorithms to be offloaded to Tesseract cores. Many other papers used general-purpose scalar/vector processors or even FPGAs as in-memory computation units.

While this traditional style of PIM architecture design helps achieving the maximal performance benefit from the PIM concept, it introduces two major challenges in seamlessly integrating PIM architectures into conventional systems in the near term. First, prior proposals require new programming models for in-memory computation units, which is often significantly different from what is used today. Main memory products with integrated full-fledged processors with new programming models may also not be available in the near future because of the associated design complexity and changes required across the hardware/software stack.

Second, prior proposals do not utilize the benefits of on-chip caches and virtual memory provided by host processors. Specifically, they offload computation to memory with no consideration of on-chip cache locality, thereby significantly degrading performance when the applications exhibit high data locality. Moreover, with respect

This chapter is originally published in Proceedings of the International Symposium on Computer Architecture, 2015 [155].

to cache coherence, most previous PIM approaches either operate on a noncacheable memory region [18–20, 27, 68, 157], insert explicit cache block flushes into software code [21, 22, 157, 159], or require invalidations of a memory region [55], all of which introduce performance overhead and/or programming burden. Although it is possible to extend host-side cache coherence protocols to main memory [160], this incurs nontrivial overhead in implementing coherence protocols inside memory and, more importantly, tightly couples memory design to host processor design. The lack of interoperability with on-chip caches is critical considering that commercial processors already integrate large last-level caches on chip (e.g., Intel Ivytown has a 37.5 MB L3 cache [6]). In addition to that, most prior approaches perform in-memory computation on physically addressed memory regions (i.e., no support for virtual memory), which inevitably sacrifices safety of all memory accesses from host processors to memory regions that can potentially be accessed by PIM.

To overcome these two major limitations, this chapter proposes to enable simple PIM operations by extending the ISA of the host processor with *PIM-enabled instructions (PEIs)*, without changing the existing programming model. We define a PEI as an instruction that can be executed either on the host processor or on the in-memory computation logic. By providing PIM capability as part of the existing programming model, conventional architectures can exploit the PIM concept with no changes in their programming interface. The approach of using PEIs also facilitates our architecture to support cache coherence and virtual memory seamlessly: existing mechanisms in place for cache coherence and virtual memory can be used without modification, unlike other approaches to PIM. We develop a hardware-based scheme that can adaptively determine the location to execute PEIs by considering data locality: a PEI can be selectively executed on the host processor (instead of on the in-memory computation logic) when large on-chip caches are beneficial for its execution.

This chapter makes the following major contributions:

- We introduce the concept of PIM-enabled instructions (PEIs), which enable simple PIM operations to be interfaced as simple ISA extensions. We show that the judicious usage of PEIs can achieve significant speedup with minimal programming effort and no changes to the existing programming model.

- We design an architecture that supports implementing PEIs as part of the host-PIM interface in order to provide the illusion that PIM operations are executed as if they were host processor instructions. Unlike previous PIM proposals, PEIs are fully interoperable with existing cache coherence and virtual memory mechanisms.

- We propose a mechanism to dynamically track locality of data accessed by PEIs and execute PEIs with high data locality on host processors instead of offloading them to memory, in order to exploit large on-chip caches.
- We evaluate our architecture using ten emerging data-intensive workloads and show significant performance improvements over conventional systems. We also show that our architecture is able to adapt to data locality at runtime, thereby outperforming PIM-only systems as well.

9.1 Potential of ISA Extensions as the PIM Interface

Past work on PIM, including early proposals [18, 20–22, 156, 157] and more recent ones [7,27,58,68], mostly relies on fully programmable computation units (e.g., general-purpose processors, programmable logic, etc.) in memory. While programmability gives the benefit of broad applicability across different workloads, such approaches may not be suitable for a near-term solution due to (1) high design effort required to integrate such complex modules into memory and (2) new programming models for in-memory computation units, which require significant changes to software code to exploit PIM.

Motivated by these difficulties, we explore possibilities of *integrating simple PIM operations by minimally extending the ISA of host processors*. Compared to PIM architectures based on fully programmable computation units, our approach improves the practicality of the PIM concept by reducing the implementation overhead of in-memory computation units and facilitating the use of existing programming models.[1] However, despite these advantages, no prior work, to our knowledge, explored the methods and their effectiveness of utilizing *simple* in-memory computation.

To evaluate the potential of introducing simple PIM operations as ISA extensions, let us consider a parallel implementation of the PageRank algorithm [137], shown in Figure 9.1, as an example. PageRank [151] is widely used in web search engines, spam detection, and citation ranking to compute the importance of nodes based on their relationships. For large graphs, the performance bottleneck of this workload is in updating `next_pagerank` of successor vertices (line 10) since it generates a very large amount of random memory accesses across the entire graph [161]. Due to its high memory intensity and computational simplicity, this operation in PageRank is a good candidate for implementation as a simple PIM operation.

[1] Due to these advantages, simple in-memory computation mechanisms are already starting to become available from the industry, e.g., the in-memory 8/16-byte arithmetic/bitwise/boolean/comparison atomics in HMC 2.0 [28].

```
1  for (v: graph.vertices) {
2    v.pagerank = 1.0 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    for (v: graph.vertices) {
8      delta = 0.85 * v.pagerank / v.out_degree;
9      for (w: v.successors) {
10       atomic w.next_pagerank += delta;
11     }
12   }
13   diff = 0.0;
14   for (v: graph.vertices) {
15     atomic diff += abs(v.next_pagerank - v.pagerank);
16     v.pagerank = v.next_pagerank;
17     v.next_pagerank = 0.15 / graph.num_vertices;
18   }
19 } while (++count < max_iteration && diff > e);
```

Figure 9.1: Pseudocode of parallel PageRank computation.

Figure 9.2 shows speedup achieved by implementing an *atomic addition* command inside memory (see Section 9.5 for our evaluation methodology). The evaluation is based on nine real-world graphs [152, 162], which have 62 K to 5 M vertices (y-axis is sorted in ascending order of the number of vertices).

We observe that employing only one simple PIM operation can improve system performance by up to 53%. The main benefit is due to the greatly reduced memory bandwidth consumption of the memory-side addition compared to its host-side counterpart. While host-side addition moves the entire cache block back and forth between memory and the host processor for each update, memory-side addition simply sends the 8-byte `delta` to memory to update `w.next_pagerank`. Assuming 64-byte cache blocks and no cache hits, the host-side addition transfers 128 bytes of data from/to off-chip memory for every single update, whereas the memory-side addition requires only 8 bytes of off-chip communication per update.

Unfortunately, memory-side addition sometimes incurs significant performance degradation as well (up to 20%). This happens when most of `next_pagerank` updates can be served by on-chip caches, in which case host-side addition does *not* consume off-chip memory bandwidth at all. In such a situation, memory-side addition degrades both performance and energy efficiency since it *always* accesses DRAM to update data

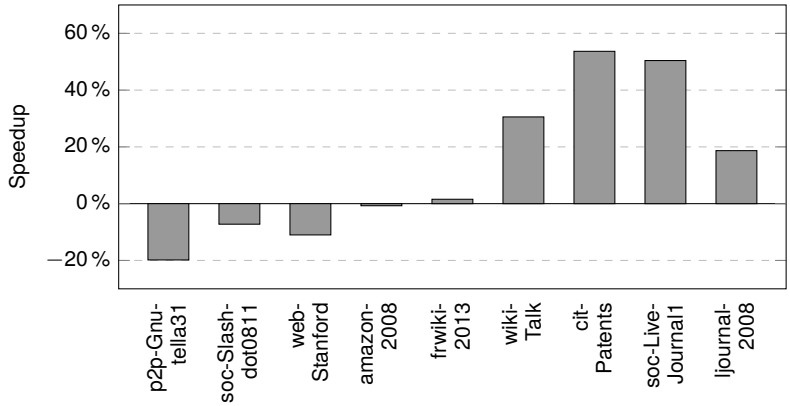


Figure 9.2: Performance improvement with an in-memory atomic addition operation used for the PageRank algorithm.

(e.g., memory-side addition causes 50x DRAM accesses over host-side addition in p2p-Gnutella31). Thus, one needs to be careful in using memory-side operations as their benefit greatly depends on the locality of data in on-chip caches.

In summary, simple PIM operations interfaced as ISA extensions have great potential for accelerating memory-intensive workloads. However, in order to maximize the effectiveness of simple PIM operations, a host processor should be smart enough in utilizing these operations (e.g., importantly, by considering data locality of applications). Based on these observations, in the following sections, we describe our new, simple PIM interface consisting of simple ISA extensions and the architectural support required to integrate such simple PIM operations into conventional systems.

9.2 PIM Abstraction

In this section, we explain our abstraction model for simple PIM operations, called PIM-enabled instructions (PEIs). Our goal is to *provide the illusion that PIM operations are executed as if they were host processor instructions.* This section describes our design choices to achieve this new PIM abstraction. Section 9.3 will describe our detailed implementation to realize this abstraction.

9.2.1 Operations

In order to integrate PIM capability into an existing ISA abstraction, PIM operations are expressed as specialized instructions (PEIs) of host processors. For example, if

the main memory supports an in-memory *atomic add* command (PIM operation), we add a PIM-enabled *atomic add* instruction (PEI) to the host processor. This facilitates effortless and gradual integration of PIM operations into existing software through replacement of normal instructions with PEIs.

When a PEI is issued by a host processor, our hardware mechanism dynamically decides the best location to execute it between memory and the host processor on a per-operation basis. Software does not provide any information to perform such a decision and is unaware of the execution location of the operation, which is determined by hardware.

For memory operands of PIM operations, we introduce an important design choice, called the *single-cache-block restriction*: the memory region accessible by a single PIM operation is limited to a single last-level cache block. Such a restriction brings at least three important benefits in terms of efficiency and practicality of PIM operations, as described below.

- **Localization:** It ensures that data accessed by a PIM operation are always bounded to a single DRAM module. This implies that PIM operations always use only vertical links (without off-chip data transfer) in transferring the target data between DRAM and in-memory computation units.[2]
- **Interoperability:** Since PIM operations and normal last-level cache accesses now have the same memory access granularity (i.e., one last-level cache block), hardware support for coherence management and virtual-to-physical address translation for PIM operations becomes greatly simplified (see Section 9.3).
- **Simplified Locality Profiling:** Locality of data accessed by PIM operations can be easily identified by utilizing the last-level cache tag array or similar structures. Such information is utilized in determining the best location to execute PEIs.

In addition to memory operands, PIM operations can also have input/output operands. An example of this is the `delta` operand of the atomic addition at line 10 of Figure 9.1. When a PIM operation is executed in main memory, its input/output operands are transferred between host processors and memory through off-chip links. The maximum size of input/output operands is restricted to the size of a last-level cache block because, if input/output operands are larger than a last-level cache block, offloading such PIM

[2]Without the single-cache-block restriction, PIM operations require special data mapping to prevent off-chip transfer between multiple DRAM modules. This comes with many limitations in that (1) it introduces significant modification to existing systems to expose the physical location of data to software and (2) the resulting design may not be easily adaptable across different main memory organizations.

operations to memory increases off-chip memory bandwidth consumption compared to host-side execution due to the single-cache-block restriction.

9.2.2 Memory Model

Coherence. Our architecture supports hardware cache coherence for PIM operations so that (1) PIM operations can access the latest versions of data even if the target data is stored in on-chip caches and (2) normal instructions can see the data modified by PIM operations. This allows programmers to mix normal instructions and PEIs in manipulating the same data without disabling caches, in contrast to many past PIM architectures.

Atomicity. Our memory model ensures atomicity between PEIs. In other words, a PEI that reads from and writes to its target cache block is not interrupted by other PEIs (possibly from other host processors) that access the same cache block. For example, if the addition at line 10 of Figure 9.1 is implemented as a PEI, hardware preserves its atomicity without any software concurrency control mechanism (e.g., locks).

On the contrary, atomicity of a PEI is not guaranteed if a normal instruction accesses the target cache block. For example, if a normal store instruction writes a value to `w.next_pagerank` (in Figure 9.1), this *normal* write may happen between reading `w.next_pagerank` and writing it inside the atomic PEI addition (line 10), thereby breaking the atomicity of the PEI. This is because, in order to support atomicity between a PEI and a normal instruction, every memory access needs to be checked, which incurs overhead even for programs that do not use PEIs.

Instead, host processors provide a PIM memory fence instruction called *pfence* to enforce memory ordering between normal instructions and PEIs. The pfence instruction blocks host processor execution until all PEIs issued before it (including those from other host processors) are completed. In Figure 9.1, a pfence instruction needs to be inserted after line 12 since normal instructions in the third loop access data modified by PEIs in the second loop (i.e., the `w.next_pagerank` fields).

It should be noted that, although pfence itself might introduce some performance overhead, its overhead can generally be amortized over numerous PEI executions. For example, the PageRank algorithm in Figure 9.1 issues one PEI per edge before each pfence, which corresponds to millions or even billions of PEIs per pfence for large real-world graphs.

Virtual Memory. PEIs use virtual addresses just as normal instructions. Supporting virtual memory for PEIs therefore does not require any modification to existing operating systems and applications.

9.2.3 Software Modification

In this chapter, we assume that programmers modify source code of target applications to utilize PEIs. This is similar to instruction set extensions in commercial processors (e.g., Intel SSE/AVX [163]), which are exploited by programmers using intrinsics and are used in many real workloads where they provide performance benefit. However, we believe that the semantics of the PEIs is simple enough (e.g., atomic add) for modern compilers to automatically recognize places to emit them without requiring hints from programmers and/or complex program analysis. Moreover, our scheme reduces the burden of compiler designers since it *automatically* determines the best location to execute each PEI between memory and the host processor, which allows compilers to be less accurate in estimating performance/energy gain of using PEIs.

9.3 Architecture

In this section, we describe our architecture that implements our PIM abstraction with minimal modification to existing systems. Our architecture is not limited to specific types of in-memory computation, but provides a *substrate* to implement simple yet general-purpose PIM operations in a practical and efficient manner.

9.3.1 Overview

Figure 9.3 gives an overview of our architecture. We choose the Hybrid Memory Cube (HMC) [24] as our baseline memory technology. An HMC consists of multiple vertical DRAM partitions called *vaults*. Each vault has its own DRAM controller placed on the logic die. Communication between host processors and HMCs is based on a packet-based abstract protocol supporting not only read/write commands, but also compound commands such as add-immediate operations, bit-masked writes, and so on [26]. Note that our architecture can be easily adopted to other memory technologies since it does not depend on properties specific to any memory technology.

The key features of our architecture are (1) to support PIM operations as part of the host processor instruction set and (2) to identify PEIs with high data locality and execute them in the host processor. To realize these, our architecture is composed of two main components. First, a PEI Computation Unit (PCU) is added into each host

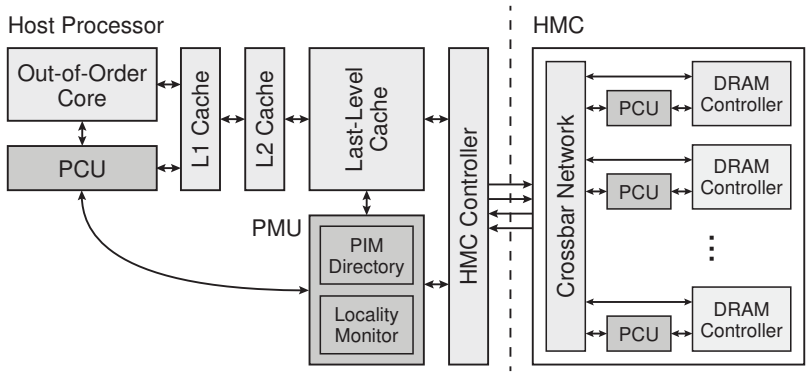


Figure 9.3: Overview of the proposed architecture.

processor and each vault to facilitate PEIs to be executed in either the host processor or main memory. Second, in order to coordinate PEI execution in different PCUs, the PEI Management Unit (PMU) is placed near the last-level cache and is shared by all host processors. In the following subsections, we explain the details of these two components and their operation sequences in host-/memory-side PEI execution.

9.3.2 PEI Computation Unit (PCU)

Architecture. A PCU is a hardware unit that executes PEIs. Each PCU is composed of computation logic and an operand buffer. The computation logic is a set of circuits for computation supported by main memory (e.g., adders). All PCUs in the system have the same computation logic so that any PEI can be executed by any PCU.

The operand buffer is a small SRAM buffer that stores information of in-flight PEIs. For each PEI, an operand buffer entry is allocated to store its type, target cache block, and input/output operands. When the operand buffer is full, future PEIs are stalled until space frees up in the buffer.

The purpose of the operand buffer is to exploit memory-level parallelism during PEI execution. In our architecture, when a PEI obtains a free operand buffer entry, the PCU immediately sends a read request for the target cache block of the PEI to memory even if the required computation logic is in use. Then, the fetched data is buffered in the operand buffer until the computation logic becomes available. As such, memory accesses from different PEIs can be overlapped by simply increasing the number of operand buffer entries. This is especially useful in our case since simple PIM operations usually underutilize the computation logic of PCUs due to the small amount of computation they generate.

Interface. A host processor controls its host-side PCU by manipulating memory-mapped registers inside it (see Section 9.3.5). Assemblers can provide pseudo-instructions for PCU control, which are translated to accesses to those memory-mapped registers, in order to abstract the memory-mapped registers away from application software. Although we choose this less invasive style of integration to avoid modification to out-of-order cores, one can add actual instructions for PCU control by modifying the cores for tighter integration.

Memory-side PCUs are interfaced with the HMC controllers using special memory commands. It is relatively easy to add such commands because communication between HMCs and HMC controllers is based on a packet-based abstract protocol, which allows the flexible addition of new commands.

9.3.3 PEI Management Unit (PMU)

In order to coordinate all PCUs in the system, the PMU performs three important tasks for PEI execution: (1) atomicity management of PEIs, (2) cache coherence management for PIM operation execution, and (3) data locality profiling for locality-aware execution of PIM operations. We explain in detail the proposed hardware structures to handle these tasks.

PIM Directory. As explained in Section 9.2.2, our architecture guarantees the atomicity of PEIs. If we consider memory-side PEI execution only, atomicity of PEIs can be maintained simply by modifying each DRAM controller inside HMCs to schedule memory accesses (including reads and writes) from a single PEI as an inseparable group. However, since our architecture executes PEIs in both host-side PCUs and memory-side PCUs, this is not enough to guarantee the atomicity of host-side PEI execution.

In order to guarantee the atomicity of both host-side and memory-side PEI execution, our architecture introduces a hardware structure that manages atomicity of PEI execution at the host side. Ideally, this structure would track *all* in-flight PEIs to ensure that each cache block has either only one writer PEI (i.e., a PEI that modifies its target cache block) or multiple reader PEIs (i.e., PEIs that only read their target cache block). However, this incurs a very large overhead since exact tracking of such information requires a fully associative table having as many entries as the total number of operand buffer entries of all PCUs in the system (which is equal to the maximum number of in-flight PEIs, as discussed in Section 9.3.2).

We develop a special hardware unit called *PIM directory* to manage atomicity of in-flight PEIs in a cost-effective manner. The key idea of the PIM directory is to

allow rare false positives in atomicity management (i.e., serialization of two PEIs with different target cache blocks) for storage overhead reductions. This does not affect the atomicity of PEIs as long as there are no false negatives (e.g., simultaneous execution of two writer PEIs with the same target cache block).[3] In order to exploit this idea, the PIM directory is organized as a direct-mapped, tag-less table indexed by XOR-folded addresses of target cache blocks. Each entry implements a reader-writer lock with four fields: (1) a readable bit, (2) a writeable bit, (3) an n-bit reader counter where $n = \lceil \log_2(\text{total number of operand buffer entries}) \rceil$, and (4) a 1-bit writer counter.

When a reader PEI arrives at the PIM directory, it is blocked until the corresponding PIM directory entry is in the readable state. After that, the entry is marked as non-writeable in order to block future writer PEIs during the reader PEI execution. At this moment, the reader PEI can be executed atomically. After the reader PEI execution, if there are no more in-flight reader PEIs to the entry, the entry is marked as writeable.

When a writer PEI arrives, it first needs to ensure that there are no in-flight writer PEIs for the corresponding PIM directory entry since atomicity allows only a single writer PEI for each cache block. Then, the entry is marked as non-readable to avoid write starvation by future reader PEIs. After that, the writer PEI waits until all in-flight reader PEIs for the PIM directory entry are completed (i.e., until the PIM directory entry becomes writeable), in order to prevent the reader PEIs from reading the cache block in the middle of the writer PEI execution. Finally, the write PEI is executed atomically and, upon its completion, the state of the entry is set to readable.

In addition to atomicity management, the PIM directory also implements the pfence instruction explained in Section 9.2.2. When a pfence instruction is issued, it waits for each PIM directory entry to become readable. This ensures that all in-flight writer PEIs issued before the pfence are completed when the pfence returns.

Cache Coherence Management. In our PIM abstraction, PEIs should be interoperable with existing cache coherence protocols. This is easily achievable for host-side PEI execution since host-side PCUs share L1 caches with their host processors. On the other hand, PEIs offloaded to main memory might read stale values of their target cache blocks if on-chip caches have modified versions of these blocks.

Our solution to this problem is simple. Due to the single-cache-block restriction, when the PMU receives a PEI, it knows exactly which cache block the PEI will access.

[3] Although too frequent false positives may incur a performance overhead due to unnecessary serialization, our evaluation shows that our mechanism achieves similar performance to its ideal version (an infinite number of entries) while incurring only a small storage overhead (see Section 9.6.6).

Thus, it simply requests back-invalidation (for writer PEIs) or back-writeback[4] (for reader PEIs) for the target cache block to the last-level cache before sending the PIM operation to memory. This ensures that neither on-chip caches nor main memory has a stale copy of the data before/after PIM operation execution, without requiring complex hardware support for extending cache coherence protocols toward the main memory side. Note that back-invalidation and back-writeback happen infrequently in practice since our architecture offloads PEIs to memory *only if* the target data is not expected to be present in on-chip caches.

Locality Monitor. One of the key features of our architecture is locality-aware PEI execution. The key idea is to decide whether to execute a PEI locally on the host or remotely in memory. This introduces a new challenge: dynamically profiling the data locality of PEIs. Fortunately, our single-cache-block restriction simplifies this problem since data locality of each PEI depends only on the locality of a *single* target cache block. With that in mind, the remaining issue is to monitor locality of the cache blocks accessed by the PEIs, which is done by the *locality monitor* in the PMU.

The locality monitor is a tag array with the same number of sets/ways as that of the last-level cache. Each entry contains a valid bit, a 10-bit partial tag (constructed by applying folded-XOR to the original tags), and replacement information bits. Each last-level cache access leads to hit promotion and/or block replacement for the corresponding locality monitor entry (as in the last-level cache).

The key difference between the locality monitor and the tag array of the last-level cache is that the former is also updated *when a PIM operation is issued to memory*. More specifically, when a PIM operation is sent to main memory, the locality monitor is updated as if there is a last-level cache access to its target cache block. By doing so, the locality of PEIs is properly monitored regardless of the location of their execution.

In our locality monitor, the data locality of a PEI can be identified by checking to see if its target cache block address hits in the locality monitor. The key idea behind this is that, if some cache blocks are accessed frequently (i.e., high locality), they are likely to be present in the locality monitor. However, according to our evaluations, if a locality monitor entry is allocated by a PIM operation, it is often too aggressive to consider it as having high locality on its first hit in the locality monitor. Therefore, we add a 1-bit ignore flag per entry to ignore the first hit to an entry allocated by a PIM operation (those allocated by last-level cache accesses do not set this ignore flag).

[4]We use the term 'back-writeback' to denote a writeback request that forces writing back the target cache block from any of the L1, L2, or L3 caches it is present in to main memory (analogous to back-invalidation in inclusive cache hierarchies).

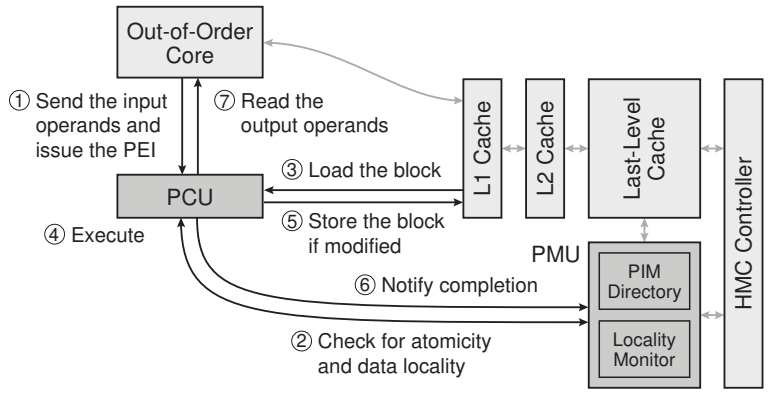


Figure 9.4: Host-side PEI execution.

9.3.4 Virtual Memory Support

Unlike many other PIM architectures, virtual memory is easily supported in our architecture since PEIs are part of the conventional ISA. When the host processor issues a PEI, it simply translates the virtual address of the target cache block by accessing its own TLB. By doing so, all PCUs and the PMU in the system handle PEIs with physical addresses only.

This scheme greatly improves the practicality of our architecture. First, it avoids the overhead of adding address translation capabilities in memory. Second, existing mechanisms for handling page faults can be used without modification because page faults are still handled only at the host processor (i.e., no in-memory page faults). Third, it does not increase TLB pressure since the single-cache-block restriction guarantees that only one TLB access is needed for each PEI just as a normal memory access.

9.3.5 PEI Execution

Host-side PEI Execution. Figure 9.4 illustrates the execution sequence of a PEI with high data locality. First, the host processor sends the input operands of the PEI to the PCU and issues it ①. The host-side PCU has a set of memory-mapped registers for control and temporary storage of input operands. Next, the host-side PCU accesses the PMU to (1) obtain a reader-writer lock for the target cache block from the PIM directory and (2) consult the locality monitor to decide the best location to execute the PEI ②. In this case, the locality monitor advises the execution of the PEI on the host-side PCU as the target block is predicted to have high data locality. After that, the host-side PCU allocates a new operand buffer entry, copies the input operands from the

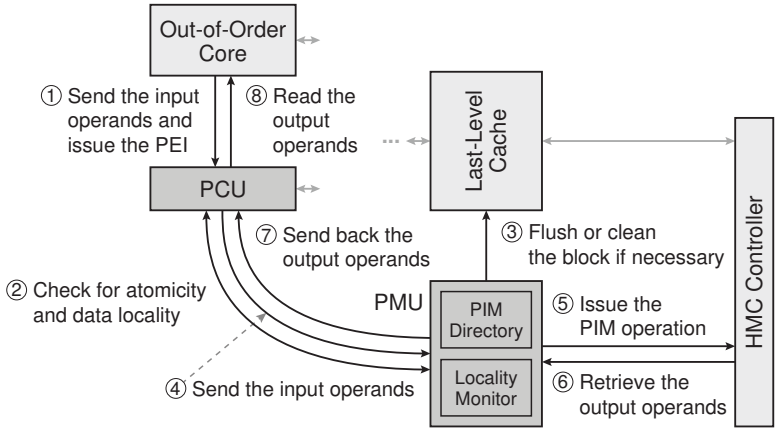


Figure 9.5: Memory-side PEI execution.

memory-mapped register to this entry, and loads the cache block from the L1 cache ③. Once the target cache block is loaded, the PCU executes the PEI ④ and initiates a store request to the L1 cache if the PEI modifies the target cache block data ⑤. When the PEI execution is complete, the host-side PCU notifies the completion to the PMU in background to release the corresponding PIM directory entry ⑥. Finally, the host processor reads the output operands through memory-mapped registers inside the PCU and the operand buffer entry is deallocated ⑦.

Memory-side PEI Execution. Figure 9.5 shows the procedure of executing a PEI in main memory. Steps ① and ② are the same as in the above case, except that the locality monitor advises the execution of the PEI in memory. While this decision is being returned to the host-side PCU, the PMU sends a back-invalidation/back-writeback signal to the last-level cache to clean any local copy of the target cache block ③. At the same time, the host-side PCU transfers the input operands stored in its memory-mapped registers to the PMU ④. Once both steps are complete, the PMU sends the PIM operation to main memory by packetizing the operation type, the target cache block address, and the input operands ⑤. Upon receiving the response from the main memory ⑥, the PMU releases the corresponding PIM directory entry and sends the output operands back to the host-side PCU ⑦ so that the host processor can read them ⑧.

9.3.6 Comparison with Active Memory Operations

In terms of PIM operation granularity, the most relevant research to ours is active memory operations (AMOs) [164, 165], in which *on-chip memory controllers* support a

limited set of simple computations. However, our approach is different from AMOs in at least three aspects. First, unlike our PIM-based architecture, computation in *on-chip memory controllers*, as opposed to in *memory*, still suffers from the off-chip bandwidth limitation, which is the bottleneck of our target applications. Second, AMOs are always executed in memory controllers, which requires cache block flushes for each AMO, thereby degrading performance compared to host-side execution under high data locality. This is not the case for our system since host-side PCUs see exactly the same memory hierarchy as the host processor. Third, the memory controller design for AMOs needs dedicated TLBs for virtual memory support, whereas our architecture achieves the same goal without such overhead by performing address translation with host processor TLBs.

9.4 Target Applications for Case Study

The primary target of our architecture is applications with large memory footprint and very large memory bandwidth consumption. This makes it difficult to use standard benchmarks for evaluation as many of them do not stress off-chip memory. Instead, we perform a case study on ten data-intensive workloads from three important domains, which are often classified as "big-data" workloads. Evaluating such big-data applications on PIM architectures is very important since they are envisioned as the principal target for PIM due to their importance, broad applicability, and enormous memory bandwidth demand [23, 27]. Note that other applications can also be accelerated under our general-purpose framework by implementing (possibly) different types of PEIs.

9.4.1 Large-Scale Graph Processing

Average Teenage Follower (ATF) [149] is an example kernel of social network analysis. It counts for each vertex the number of its teenage followers by iterating over all teenager vertices and incrementing the 'follower' counters of their successor vertices. Since this generates a very large amount of random memory accesses over the entire graph (pointer chasing over edges), we implement *8-byte atomic integer increment* as a PIM operation to accelerate it.

Breadth-First Search (BFS) [149] is a graph traversal algorithm, which visits vertices closer to a given source vertex first. Our implementation is based on the parallel level-synchronous BFS algorithm [149, 166], where each vertex maintains a 'level' field to track the progress of traversal. Since each vertex updates the level fields of its neighbors by a min function, we implement *8-byte atomic integer min* as a PIM

operation, which accepts an 8-byte input operand that replaces the target 8-byte word in memory if the input operand is smaller than the target word in memory.

PageRank (PR) [137] is a well-known algorithm that calculates the importance of vertices in a graph (see Figure 9.1). We implement *double-precision floating point addition* as a PIM operation for this application as discussed in Section 9.1.

Single-Source Shortest Path (SP) [149] finds the shortest path from a given source vertex to other vertices in a graph. Our application uses the parallel Bellman-Ford algorithm, which repeatedly iterates over vertices with distance changes and relaxes their outgoing edges with min functions. Therefore, our implementation uses the *atomic integer min operation* that we already discussed for BFS.

Weakly Connected Components (WCC) [167] identifies weakly connected components of a graph (a set of vertices that are reachable from each other when edge direction is ignored). The algorithm assigns a unique label for each vertex and collapses the labels of connected vertices into the smallest one by propagating labels through edges. Thus, this workload can also benefit from the *atomic integer min* operation.

9.4.2 In-Memory Data Analytics

Hash Join (HJ) [168] joins two relations from an in-memory database by building a hash table with one relation and probing it with keys from the other. For this application, we implement a PIM operation for hash table probing, which checks keys in a given bucket for a match and returns the match result and the next bucket address. The host processor issues this PIM operation for the next bucket by performing pointer chasing with the returned next bucket address. We also modify the software to unroll multiple hash table lookups for different rows to be interleaved in one loop iteration. This allows out-of-order cores to overlap the execution of multiple PIM operations with the use of out-of-order execution.

Histogram (HG) builds a histogram with 256 bins from 32-bit integer data. In order to reduce memory bandwidth consumption of reading the input data, we implement a PIM operation that calculates the bin indexes of data in memory by shifting each 4-byte word in a 64-byte cache block with a given shift amount, truncating each word into a 1-byte value, and returning all 16 of them as a 16-byte output operand.

Radix Partitioning (RP) [168] is a data partitioning algorithm for in-memory databases, which acts as a preprocessing step for many database operations. Since it internally builds a histogram of data before partitioning the data, it can be accelerated by using the PIM operation for HG. However, unlike HG where input data is read only once in a streaming manner, RP accesses the original data again after building the histogram

Table 9.1: Summary of Supported PIM Operations

Operation	R	W	Input	Output	Applications
8-byte integer increment	O	O	0 bytes	0 bytes	AT
8-byte integer min	O	O	8 bytes	0 bytes	BFS, SP, WCC
Floating-point add	O	O	8 bytes	0 bytes	PR
Hash table probing	O	X	8 bytes	9 bytes	HJ
Histogram bin index	O	X	1 byte	16 bytes	HG, RP
Euclidean distance	O	X	64 bytes	4 bytes	SC
Dot product	O	X	32 bytes	8 bytes	SVM

to move the data to corresponding partitions. We simulate a usage scenario where this algorithm is applied to given input data 100 times, which resembles access patterns of database servers continuously receiving queries to the same relation.

9.4.3 Machine Learning and Data Mining

Streamcluster (SC) [104] is an online clustering algorithm for n-dimensional data. The bottleneck of this algorithm is in computing the Euclidean distance of two points. To accelerate this computation, we implement a PIM operation that computes the distance between two 16-dimensional single-precision floating-point vectors, one stored in its target cache block (A) and the other passed as its input operand (B). Since the application uses this kernel to calculate distance from *few* cluster centers to *many* data points, we use the PIM operation by passing a cluster center as B and a data point as A, to preserve the locality of the cluster centers.

Support Vector Machine Recursive Feature Elimination (SVM) [169] selects the best set of features that describe the data in support vector machine (SVM) classification. It is extensively used in finding a compact set of genes that are correlated with disease. The SVM kernel inside it heavily computes dot products between a single hyperplane vector (w) and a very large number of input vectors (x). Thus, we implement a PIM operation that computes the dot product of two 4-dimensional double-precision floating-point vectors, similar to the PIM operation for SC.

9.4.4 Operation Summary

Table 9.1 summarizes the PIM operations implemented for our target workloads. It also shows reader/writer flags (e.g., if the 'W' column of an operation is marked as 'O', it

Table 9.2: Baseline Simulation Configuration

Component	Configuration
Core	16 out-of-order cores, 4 GHz, 4-issue
L1 I/D-Cache	Private, 32 KB, 4/8-way, 64 B blocks, 16 MSHRs
L2 Cache	Private, 256 KB, 8-way, 64 B blocks, 16 MSHRs
L3 Cache	Shared, 16 MB, 16-way, 64 B blocks, 64 MSHRs
On-Chip Network	Crossbar, 2 GHz, 144-bit links
Main Memory	32 GB, 8 HMCs, daisy-chain (80 GB/s full-duplex)
HMC	4 GB, 16 vaults, 256 DRAM banks [26]
– DRAM	FR-FCFS, tCL = tRCD = tRP = 13.75 ns [116]
– Vertical Links	64 TSVs per vault with 2 Gb/s signaling rate [24]

indicates that the operation modifies the target cache block) and the sizes of input/output operands. All these operations are supported by both the host-side and the memory-side PCUs in our system.

9.5 Evaluation Methodology

9.5.1 Simulation Configuration

We use an in-house x86-64 simulator whose frontend is based on Pin [95]. Our simulator models microarchitectural components in a cycle-level manner, including out-of-order cores, multi-bank caches with MSHRs, on-chip crossbar network, the MESI cache coherence protocol, off-chip links of HMCs, and DRAM controllers inside HMCs. Table 9.2 summarizes the configuration for the baseline system used in our evaluations, which consists of 16 out-of-order cores, a three-level inclusive cache hierarchy, and 32 GB main memory based on HMCs.

In our system, each PCU has a single-issue computation logic and a four-entry operand buffer (512 bytes). Thus, a PCU can issue memory requests of up to four in-flight PEIs in parallel but executes each PEI serially. We set the clock frequency of host-side and memory-side PCUs to 4 GHz and 2 GHz, respectively. Our system has 16 host-side PCUs (one per host processor) and 128 memory-side PCUs (one per vault).

Within the PMU design, the PIM directory is implemented as a tag-less four-bank SRAM array that can keep track of 2048 reader-writer locks. The storage overhead of the PIM directory is 3.25 KB, or 13 bits per entry (a readable bit, a writeable bit, a

Table 9.3: Input Sets of Evaluated Applications

Application	Input Sets (Small/Medium/Large)
ATF, BFS, PR, SP, WCC	soc-Slashdot0811 (77 K vertices, 905 K edges) [162] / frwiki-2013 (1.3 M vertices, 34 M edges) [152] / soc-LiveJournal1 (4.8 M vertices, 69 M edges) [162]
HJ	R = 128 K/1 M/128 M rows, S = 128 M rows
HG	$10^6/10^7/10^8$ 32-bit integers
RP	128 K/1 M/128 M rows
SC	4 K/64 K/1 M of 32/128/128-dimensional points [104]
SVM	50/130/253 instances from Ovarian cancer dataset [169]

10-bit[5] reader counter and a 1-bit writer counter). The locality monitor has 16,384 sets and 16 ways, which is the same as the organization of the tag array of the L3 cache. Since each entry consists of a valid bit, a 10-bit partial tag, 4-bit LRU information, and a 1-bit ignore flag, the storage overhead of the locality monitor is 512 KB (3.1% of the last-level cache capacity). The access latency of the PIM directory and the locality monitor is set to two and three CPU cycles, respectively, based on CACTI 6.5 [13].

9.5.2 Workloads

We simulate ten real-world applications presented in Section 9.4 for our evaluations. To analyze the impact of data locality, we use three input sets for each workload as shown in Table 9.3. All workloads are simulated for two billion instructions after skipping their initialization phases.

9.6 Evaluation Results

In this section, we evaluate four different system configurations described below. Unless otherwise stated, all results are normalized to Ideal-Host.

- **Host-Only** executes all PEIs on host-side PCUs only. This disables the locality monitor of the PMU.
- **PIM-Only** executes all PEIs on memory-side PCUs only. This also disables the locality monitor.

[5]Since our system has 576 (= $16 \times 4 + 128 \times 4$) operand buffers in total, at least 10 bits are needed to safely track the number of in-flight PEIs.

- **Ideal-Host** is the same as Host-Only except that its PIM directory is infinitely large and can be accessed in zero cycles. This configuration represents *an idealized conventional architecture*. In this configuration, our PEIs are implemented as normal host processor instructions since atomicity of operations is preserved exactly without incurring any cost.

- **Locality-Aware** executes PEIs on both host-side and memory-side PCUs based on our locality monitor.

9.6.1 Performance

Figure 9.6 presents a performance comparison of the four systems under different workload input set sizes. All results are normalized to Ideal-Host. The last bars labeled as 'GM' indicate geometric mean across 10 workloads.

First, we confirm that the effectiveness of simple PIM operations highly depends on data locality of applications. For large inputs, PIM-Only achieves 44% speedup over Ideal-Host since using PIM operations better utilizes the vertical DRAM bandwidth inside HMCs (2x the internal bandwidth utilization on average). However, for small inputs, PIM-Only degrades average performance by 20% because PIM operations *always* access DRAM even though the data set comfortably fits in on-chip caches (causing 17x the DRAM accesses on average).

In order to analyze the impact of PIM operations on off-chip bandwidth consumption, Figure 9.7 shows the total amount of off-chip transfers in Host-Only and PIM-Only, normalized to Ideal-Host. For large inputs, PIM-Only greatly reduces off-chip bandwidth consumption by performing computation in memory and bringing *only* the necessary results to the host processor.[6] However, using PIM operations for small inputs, which usually fit in the large on-chip caches, greatly increases the off-chip bandwidth consumption (up to 502x in SC).

Our architecture exploits this trade-off between Host-Only and PIM-Only execution by adapting to the data locality of applications. As shown in Figure 9.6, our Locality-Aware system provides the speedup of PIM-Only in workloads with large inputs (47% improvement over Host-Only) by offloading 79% of PEIs to memory-side PCUs. At the same time, our proposal maintains the performance of Host-Only in workloads with small inputs (32% improvement over PIM-Only) by executing 86% (96% if HG is excluded) of PEIs on host-side PCUs.

[6] An exception to this trend is observed in SC and SVM with large inputs, which will be discussed in Section 9.6.4.

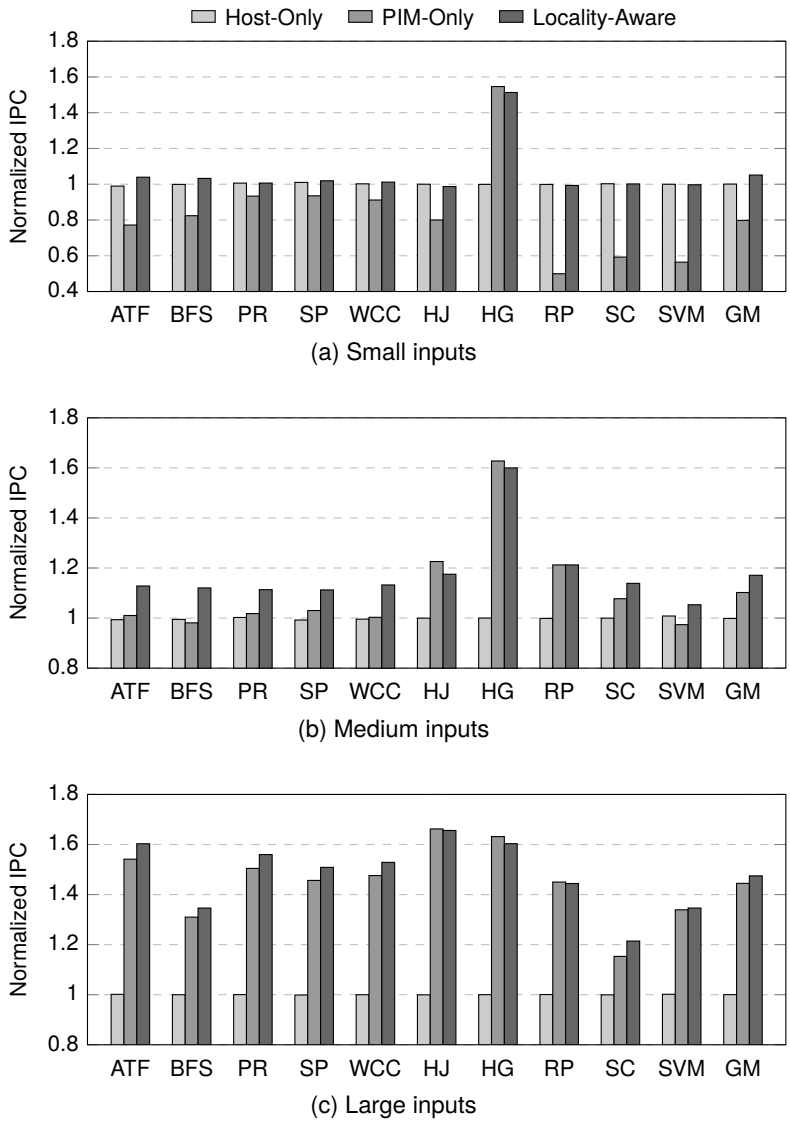


Figure 9.6: Speedup comparison under different input sizes.

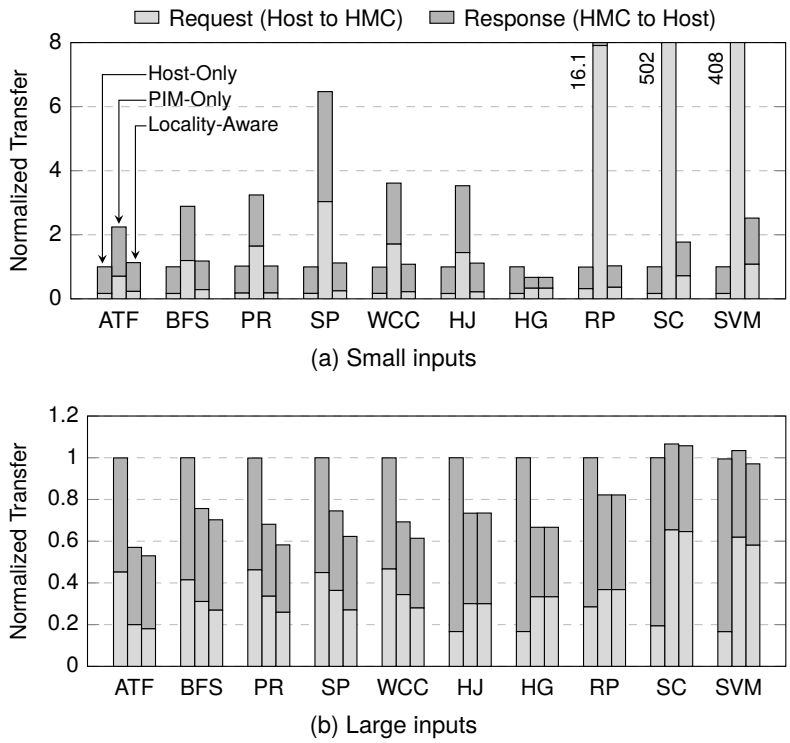


Figure 9.7: Normalized amount of off-chip transfer.

More importantly, Locality-Aware often outperforms *both* Host-Only and PIM-Only by simultaneously utilizing host-side and memory-side PCUs for PEI execution. This is especially noticeable in graph processing workloads with medium inputs (i.e., ATF, BFS, PR, SP, and WCC in Figure 9.6b), where Locality-Aware achieves 12% and 11% speedup over Host-Only and PIM-Only, respectively. The major reason is that, in those workloads, vertices have different data locality according to the shape of the graph. For example, in the PageRank algorithm shown in Figure 9.1, vertices with many incoming edges (called high-degree vertices) receive more updates than those with few incoming edges since the atomic increment at line 10 is propagated through outgoing edges. Such characteristics play an important role in social network graphs like the ones used in our evaluations since they are known to show a large variation in the number of edges per vertex (often referred to as power-law degree distribution property) [153]. In this context, Locality-Aware provides the capability of automatically optimizing computation for high-degree and low-degree vertices separately without complicating software programming.

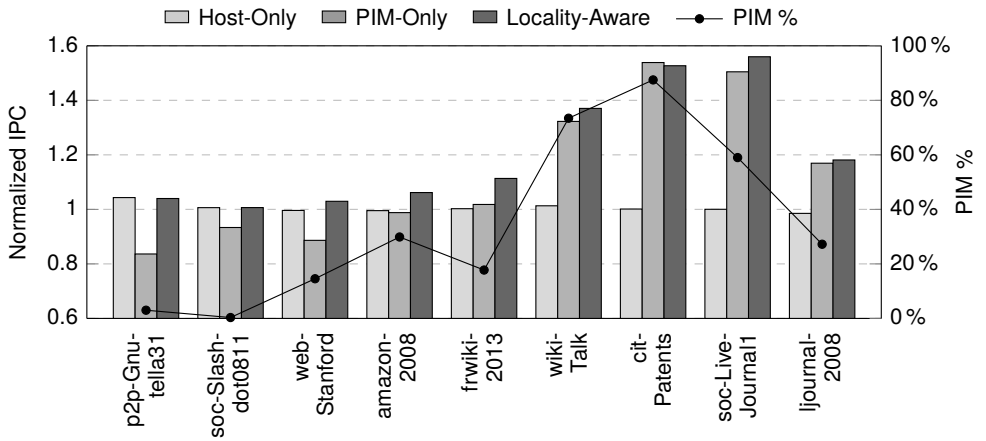


Figure 9.8: PageRank performance with different graph sizes.

9.6.2 Sensitivity to Input Size

Figure 9.8 compares the performance of Host-Only, PIM-Only, and Locality-Aware using the PageRank workload with nine input graphs. We use the same input graphs as in Figure 9.2. Graphs are sorted based on their number of vertices: graphs with larger numbers of vertices appear toward the right side of the figure. Figure 9.8 also depicts the fraction of PEIs executed on memory-side PCUs (denoted as 'PIM %').

Most notably, Locality-Aware gradually shifts from host-side execution to memory-side execution as input size grows from left to right in the figure. For example, only 0.3% of the PEIs are executed on memory-side PCUs in soc-Slashdot0811, while this ratio reaches up to 87% in cit-Patents. This indicates the adaptivity of our architecture to a wide range of data locality behavior and different input sets. Note that the locality of a graph may not be always proportional to its vertex count because it also depends on how vertices are connected with each other, which determines the memory access patterns during neighbor traversal (e.g., graphs with many high-degree nodes will have higher data locality than those with uniform distribution of edges).

We confirm that our locality monitoring scheme facilitates the use of both host-side and memory-side execution in a robust manner. For example, in amazon-2008 and frwiki-2013, which are medium-size input sets (and thus do not fully fit in on-chip caches), our technique enables a sizeable fraction of PEIs to be executed on the memory side, yet a sizeable fraction is also executed on the host side. This adaptive behavior of PEI execution shows the importance of hardware-based schemes for locality-aware

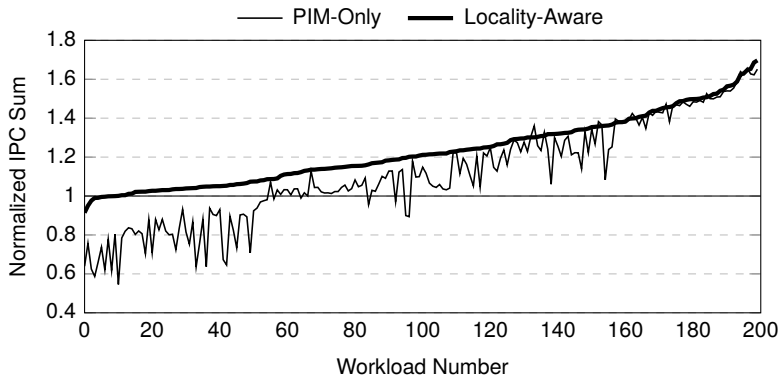


Figure 9.9: Performance comparison using randomly picked multiprogrammed workloads (normalized to Host-Only).

PIM execution, as fine-grained (per-cache-block) information of data locality cannot be easily obtained with software-only approaches.

9.6.3 Multiprogrammed Workloads

To further analyze the benefit of our dynamic mechanism for locality-aware PIM execution, we evaluate our architecture with 200 multiprogrammed workloads. Each workload is constructed by randomly picking two target applications, each of which is set to spawn eight threads. We choose their input size uniformly at random from six possible combinations (small-small, medium-medium, large-large, small-medium, medium-large, and small-large). We use the sum of IPCs as a performance metric since most of our target applications are server workloads, which are throughput-oriented.

Figure 9.9 shows the performance of Locality-Aware and PIM-Only normalized to Host-Only, showing that our locality-aware architecture performs better than both Host-Only and PIM-Only execution for an overwhelming majority of the workloads. Hence, our architecture effectively selects the best place to execute PEIs even if applications with very different locality behavior are mixed. This is an important contribution since, without hardware-based locality monitoring mechanisms like ours, it is infeasible or very difficult for the software to determine where to execute a PEI, in the presence of multiple workloads scheduled at runtime. In many systems, a diverse spectrum of applications or workloads are run together and their locality behavior changes dynamically at a fine granularity. As such, it is critical that a technique like ours can dynamically profile the locality behavior and adapt PEI execution accordingly.

9.6.4 Balanced Dispatch: Idea and Evaluation

As shown in Figure 9.7, most of the speedup achieved by PIM-Only comes from the reduction in memory bandwidth consumption. However, we observe an intriguing behavior for some workloads: PIM-Only outperforms Host-Only in SC and SVM with large inputs even though it *increases* off-chip bandwidth consumption. This is because PIM-Only shows better balance between request and response bandwidth consumption (note that HMCs have separate off-chip links for requests and responses). For example, in SC, Host-Only reads 64-byte data per PEI, while PIM-Only sends 64-byte data to memory per PEI (see Table 9.1). Although the two consume nearly the same amount of memory bandwidth in total, the latter performs better because these two applications are read-dominated, which makes response bandwidth more performance-critical than request bandwidth.[7]

Leveraging this observation, we devise a simple idea called *balanced dispatch*, which relies on the host-side PEI execution capability of our architecture. In this scheme, PEIs are forced to be executed on host-side PCUs regardless of their locality if doing so achieves a better balance between request and response bandwidth. For this purpose, two counters, C_{req} and C_{res}, are added to the HMC controller to accumulate the total number of flits transferred through the request and response links, respectively (the counters are halved every 10 µs to calculate the exponential moving average of off-chip traffic). When a PEI misses in the locality monitor, if C_{res} is greater than C_{req} (i.e., higher average response bandwidth consumption), our scheme chooses the one that consumes less response bandwidth between host-side and memory-side execution of that PEI. Similarly, if C_{req} is greater than C_{res}, the execution location with less request bandwidth consumption is chosen.

As shown in Figure 9.10, balanced dispatch further improves the performance of our architecture by up to 25%. We believe that this idea can be generalized to other systems with separate request/response channels such as buffer-on-board memory systems [170] (e.g., Intel SMB [171], IBM Centaur [172], etc.), the evaluation of which is left for our future work.

9.6.5 Design Space Exploration for PCUs

Operand Buffer Size. Figure 9.11a shows the performance sensitivity of Locality-Aware to the operand buffer size in each PCU. The results are averaged over all

[7] To be more specific, a memory read consumes 16/80 bytes of request/response bandwidth and a memory write consumes 80 bytes of request bandwidth in our configuration. Thus, if an application is read-dominated, the response bandwidth is likely to be saturated first.

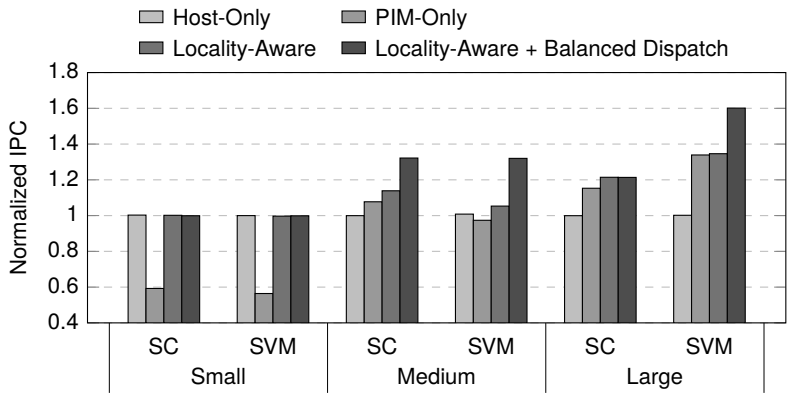


Figure 9.10: Performance improvement of balanced dispatch.

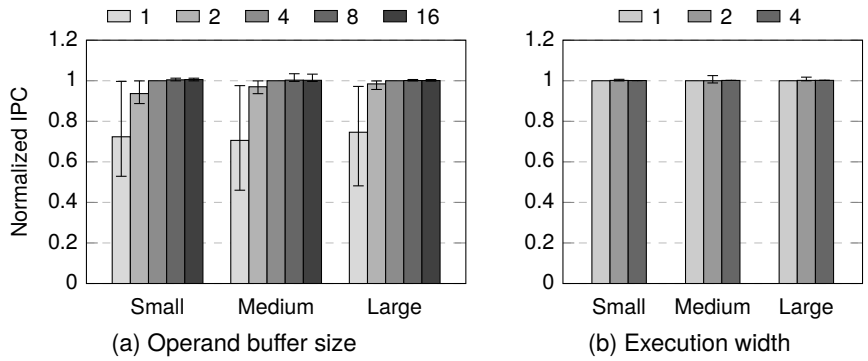


Figure 9.11: Performance sensitivity to different PCU designs.

applications and then normalized to the default configuration (four entries). Error bars show the minimum and the maximum values.

According to the results, incorporating a four-entry operand buffer per PCU (i.e., 576 in-flight PEIs) enables the PCU to exploit the maximum level of memory-level parallelism across PEIs, thereby improving the system performance by more than 30% compared to the one with a single-entry operand buffer. Having more than four operand buffer entries per PCU does not yield a noticeable difference in performance due to the saturation of instruction-level parallelism across PEIs.

Execution Width. Figure 9.11b depicts the impact of PCU execution width on the performance of Locality-Aware. As shown in the figure, increasing the issue width

of the computation logic shows a negligible effect since execution time of a PEI is dominated by memory access latency.

9.6.6 Performance Overhead of the PMU

Compared to an ideal PMU with infinite storage, our PMU design can potentially degrade performance in three ways. First, the limited PIM directory size unnecessarily serializes two PEIs with different target cache blocks if the two cache blocks happen to be mapped to the same PIM directory entry. Second, partial tags of the locality monitor could potentially report false data locality if two cache blocks in a set have the same partial tag. Third, access latency of the PIM directory and the locality monitor delays PEI execution.

Fortunately, we observe that these sources of potential performance degradation have a negligible impact on system performance. According to our evaluations, idealizing the PIM directory and the locality monitor (with unlimited storage and zero latency) improves the performance of our architecture by only 0.13% and 0.31%, respectively. Therefore, we conclude that our PMU design supports atomicity and locality-aware execution of PEIs with negligible performance overhead.

9.6.7 Energy, Area, and Thermal Issues

Figure 9.12 shows the energy consumption of the memory hierarchy in Host-Only, PIM-Only, and Locality-Aware, normalized to that of Ideal-Host. We use CACTI 6.5 [13] to model energy consumption of on-chip caches, the PIM directory, and the locality monitor. The energy consumption of 3D-stacked DRAM, DRAM controllers, and off-chip links of an HMC is modeled by CACTI-3DD [126], McPAT 1.2 [127], and an energy model from previous work [116], respectively. The energy consumption of the PCUs is derived from synthesis results of our RTL implementation of computation logic and operand buffers based on the Synopsys DesignWare Library [173].

Among the three configurations, Locality-Aware consumes the lowest energy across all input sizes. For small inputs, it minimizes DRAM accesses by executing most of the PEIs at host-side PCUs, unlike PIM-Only, which increases energy consumption of off-chip links and DRAM by 36% and 116%, respectively. For large inputs, Locality-Aware saves energy over Host-Only due to the reduction in off-chip traffic and execution time. Hence, we conclude that Locality-Aware enables energy benefits over both Host-Only and PIM-Only due to its ability to adapt PEI execution to data locality.

Figure 9.12 also clearly indicates that our scheme introduces negligible energy overhead in existing systems. Specifically, the memory-side PCUs contribute only 1.4%

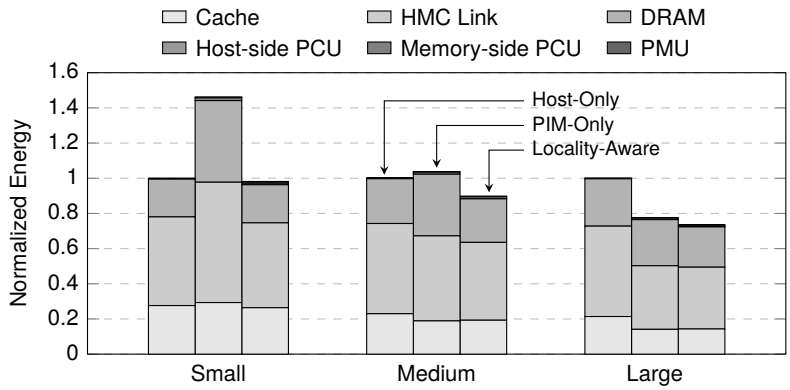


Figure 9.12: Energy consumption of memory hierarchy.

of the energy consumption of HMCs. This implies that integration of such simple PIM operations into memory likely has a negligible impact on peak temperature, which is one of the important issues in 3D-stacked memory design from a practical viewpoint [65, 174]. Finally, our synthesis results and CACTI-3DD estimate the area overhead of memory-side PCUs to be only 1.85% of the logic die area, assuming that a DRAM die and a logic die occupy the same area.

9.7 Summary

We proposed PIM-enabled instructions (PEIs), a practical model for processing-in-memory and its hardware implementation, which is compatible with existing cache coherence and virtual memory mechanisms. The key idea is to express PIM operations by extending the ISA of the host processor with PEIs. This greatly improves the practicality of the PIM concept by (1) seamlessly utilizing the sequential programming model for in-memory computation, (2) simplifying the hardware support for interoperability with existing cache coherence and virtual memory mechanisms, and (3) minimizing area, power, and thermal overheads of implementing computation units inside memory. Importantly, our architecture is also capable of dynamically optimizing PEI execution according to the data locality of applications and PEIs. Our extensive evaluation results using emerging data-intensive workloads showed that our architecture combines the best part of conventional architectures and simple PIM operations in terms of both performance and energy consumption while minimizing the overhead of in-memory computation units and management structures for PEI execution. We conclude that our PEI abstraction and its implementation provide a practical approach to realizing

high-performance and energy-efficient integration of in-memory computation capability into commodity computer systems in the near future.

Chapter 10

Aggregation-in-Memory

In the previous chapters, we presented PIM systems that can improve the performance of memory-intensive workloads by reducing off-chip data transfer. On top of that, this chapter goes one step further by *optimizing PIM architectures towards higher energy efficiency of main memory*. Although the PIM concept itself gives the performance and energy efficiency benefits to existing architectures by reducing the amount of data transfer, previous PIM proposals are suboptimal in terms of energy efficiency because they are unaware of the DRAM energy characteristics (e.g., row activation energy). Also, other than PIM-enabled instructions, most of the previous approaches always offload the target computation to memory regardless of the data locality, which can increase the energy consumption of main memory compared to conventional systems based on large on-chip caches.

To this end, we propose *Aggregation-in-Memory (AIM)*, a new PIM architecture designed for energy efficiency. AIM targets *aggregation operations*, which are defined as commutative and associative read-modify-write operations to memory-resident data. For instance, adding a given value, which we call *aggregation operand*, to data stored in memory belongs to aggregation operations. Such a type of operations is extensively used in many important data-intensive applications, including graph analysis, machine learning, histogram computation, and so on. In this chapter, we argue that offloading

This chapter is originally published in ACM Transactions on Architecture and Code Optimization, 2016 [175].

aggregation operations to memory greatly improves the energy efficiency of main memory in the following three ways:

- *Fewer main memory accesses*: While conventional architectures issue one read and one write to memory per aggregation, in-memory aggregation sends only one aggregation operation to memory.

- *Fewer row buffer misses*: An in-memory aggregation operation is processed as a single DRAM command, and thus, requires at most one row activation. On the other hand, conventional systems issue up to two row activations per aggregation since writeback caches hide the temporal locality between the read and the write of an aggregation operation from DRAM row buffers.

- *Fewer bit-flips on off-chip channels*: In-memory aggregation sends only the aggregation operands, which usually have much smaller values (i.e., lower entropy in bits) than the target data, to memory.

However, the aforementioned benefits diminish when the target data of aggregation operations can be served by on-chip caches, in which case conventional architectures do not access main memory at all. Hence, we develop a host architecture that can *adaptively* exploit in-memory aggregation by considering data locality of applications at runtime. Our hardware mechanism, called *cache-conscious aggregation*, coalesces multiple aggregation operations to the same data into a single in-memory aggregation operation by utilizing on-chip caches. This further reduces main memory accesses by introducing locality-awareness into PIM execution, which performs even better than PIM-enabled instructions.

Moreover, AIM facilitates near-term adoption of PIM into existing systems. First, aggregation operations in AIM have the identical interface to the equivalent host processor instructions including full support for cache coherence and virtual memory, which realizes an intuitive programming model for PIM. Second, since a tiny ALU would suffice to support aggregation operations, in-memory aggregation can be implemented either on the existing logic die of 3D-stacked DRAM or on the DRAM die of commodity DDRx modules, thereby providing more options for practical PIM implementation.

This chapter makes the following contributions:

- We propose *aggregation-in-memory (AIM)* for near-term adoption of energy-efficient PIM. AIM implements aggregation capability into every level of the memory hierarchy (e.g., on-chip caches, memory controllers, and main memory)

and coordinates it in a locality-aware manner, which we call *processing-in-memory-hierarchy*.

- We develop a simple programming model and compiler support for AIM, which allow existing software to automatically exploit our mechanism with no modifications.
- We show that AIM improves the energy efficiency of DRAM by reducing off-chip data transfer, improving row buffer locality, and minimizing bit-flips on memory channels. In contrast, existing PIM proposals usually provide only the first benefit under scarce data locality.
- We quantitatively analyze the effectiveness of AIM using ten important data-intensive workloads and show that AIM greatly improves energy efficiency and performance over both conventional systems and PIM-only systems.

10.1 Motivation

10.1.1 Rethinking PIM for Energy Efficiency

Many recent PIM architectures reduce the energy consumption of the memory hierarchy by reducing off-chip traffic and/or improving system performance [27, 58, 60, 61, 155]. However, we argue that they are not fully optimized yet for energy efficiency of main memory in the following two aspects.

First, while moving computation to memory reduces off-chip I/O energy consumption by avoiding off-chip data transfer, it does not necessarily improve the energy efficiency of DRAM itself. In particular, simply performing computation in memory does not improve row buffer locality, which is a key factor in determining the DRAM energy efficiency (i.e., row activation energy) [130]. Therefore, there is a plenty of room for further energy optimization in PIM system design.

Second, if the target applications exhibit enough data locality, utilizing PIM may even increase the *internal* energy consumption of main memory (e.g., row activation energy, read/write energy, etc.) compared to conventional architectures. This is because, contrary to the traditional memory hierarchy with large on-chip caches, in-memory computation units are usually equipped with a shallow cache hierarchy, thereby providing limited ability to exploit data locality for reducing internal main memory accesses. What is worse, data locality information of an application is often available only at runtime due to its input and/or system load dependence, which implies that static decision of whether to use PIM or not is impractical in many cases. Thus, PIM architectures should

```
1  for (v: graph.vertices) {
2    v.pagerank = 1 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    diff = 0;
8    for (v: graph.vertices) {
9      value = 0.85 * v.pagerank / v.out_degree;
10     for (w: v.successors) {
11       w.next_pagerank += value;
12     }
13   }
14   for (v: graph.vertices) {
15     diff += abs(v.next_pagerank - v.pagerank);
16     v.pagerank = v.next_pagerank;
17     v.next_pagerank = 0.15 / graph.num_vertices;
18   }
19 } while (diff > e && ++count < max_iteration);
```

Figure 10.1: PageRank algorithm.

be able to dynamically adapt to data locality in order to be robust against dynamic characteristics of workloads.

10.1.2 Aggregation as PIM Operations

The goal of this work is to maximize the energy efficiency benefit of PIM while minimizing the implementation cost. Thus, it is favorable to choose the computation that is difficult for conventional architectures to handle in an energy-efficient manner but at the same time computationally simple.

One type of computation that satisfies these two criteria is *aggregation operations* over a large memory region. Formally, we define an aggregation operation $v *\!\leftarrow x$ as a read-modify-write operation that computes $v * x$ for a commutative and associative operation '$*$' and stores its value back to v. For example, an increment-by-one operation can be expressed as an aggregation operation $v +\!\leftarrow 1$. Aggregation operations are importantly used in many data-intensive applications (see Section 10.5). A representative example of them is the PageRank algorithm [2, 137, 149, 151], which is widely used in web search engines and citation ranking. For large real-world graphs with millions or billions of vertices, the bottleneck of the algorithm is in updating `next_pagerank` of neighbor vertices (at line 11 of Figure 10.1), which belongs to an aggregation operation.

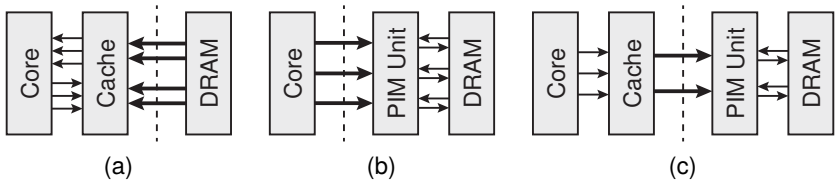


Figure 10.2: Performing aggregation in (a) conventional, (b) PIM-only, and (c) AIM systems. Each arrow indicates one access.

This is because neighbor traversal randomly accesses the entire set of vertices with very small amounts of computation, and thus, demands very high memory bandwidth [2, 7].

Despite the simplicity of aggregation operations, the conventional memory hierarchy is not optimized for processing them in an energy-efficient manner due to two reasons. First, one aggregation operation requires two memory accesses (read and write), which eventually become two main memory accesses under low data locality. Second and more importantly, one aggregation operation often incurs *two DRAM row activations*, one for read and the other for write, since on-chip caches postpone the write of the operation until the cache block eviction, which hides temporal locality between the read and the write from the row buffer.

Performing aggregation in memory solves the aforementioned inefficiencies of conventional architectures. Instead of transferring the target data back and forth between the host processor and main memory, in-memory aggregation simply sends the aggregation operand to main memory and lets the memory perform the aggregation *inside* it. This reduces the number of main memory accesses (as indicated by thick arrows in Figure 10.2b) under low data locality. Moreover, since in-memory aggregation is implemented as a single DRAM command, an aggregation operation incurs at most *one* row activation, instead of two in conventional architectures.

In order to fully exploit these benefits of in-memory aggregation, it is very important to design an intelligent host architecture as motivated previously. In particular, when the target data is stored in on-chip caches, executing all aggregation operations in memory not only increases main memory accesses but also generates extra cache traffic to flush any stale copy of the target data in on-chip caches before issuing in-memory aggregation. Ideally, the host architecture should be able to adaptively utilize both on-chip caches and in-memory aggregation capability in a way to minimize DRAM accesses, as conceptually shown in Figure 10.2c. This motivates the design of our PIM system called *Aggregation-in-Memory (AIM)*.

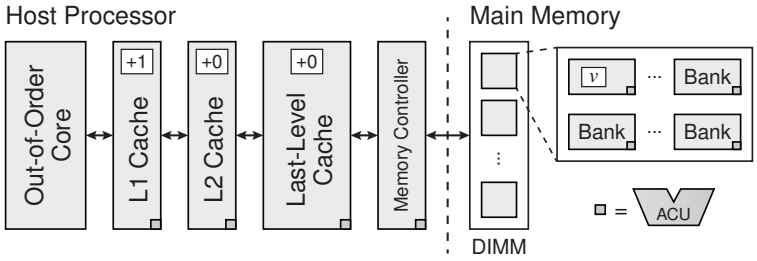


Figure 10.3: Overview of the AIM Architecture.

10.2 Architecture

10.2.1 Overview

Figure 10.3 gives an overview of the AIM architecture. It also conceptually exemplifies the contents of cache blocks (white boxes in the caches) after executing an aggregation operation $v +\leftarrow 1$ to an 8-byte integer v stored in main memory (white box in the main memory bank). Although one cache block generally contains multiple 8-byte values, other parts of the block are omitted from the figure for better visibility.

Programming Model (Section 10.2.2). AIM interfaces aggregation operations to software as cache-coherent, virtually-addressed host processor instructions. When the host processor issues an aggregation instruction, our hardware mechanism performs in-memory aggregation for the target data behind the scenes. This simplifies the programming model of AIM because the interface of in-memory aggregation is *identical* to the normal instruction for the equivalent operation.

Cache-Conscious Aggregation (Section 10.2.3). AIM avoids main memory accesses for aggregation operations with high data locality by *coalescing aggregation operations to the same data at on-chip caches*. The key idea is to store only the aggregation operands in caches without fetching the original data (e.g., '+1' in Figure 10.3). Cache-resident aggregation operands are later merged with the original data in main memory by performing in-memory aggregation.[1] To support aggregation coalescing in caches, all caches are equipped with *aggregation computation unit (ACU)*, which is a tiny ALU for aggregation.

[1]While this may change the execution order of aggregation operations, it does not affect the program semantics because of the commutativity and associativity of aggregation operations.

In-Memory Aggregation (Section 10.2.5). We implement in-memory aggregation by adding one ACU per DRAM bank. At the protocol level, in-memory aggregation is interfaced as a special write command of DRAM because both aggregation commands and write commands send cache-block-sized data to DRAM. Note that, although this work assumes commodity DDR3-based main memory (which eases the near-future integration of our idea into existing systems), AIM can also be implemented based on 3D-stacked DRAM (e.g., Hybrid Memory Cube [28]), similar to other recent PIM proposals.

To support in-memory aggregation, we introduce two modifications to the memory controllers. First, we add one ACU per memory controller to correctly handle data dependence between read commands and in-flight aggregation commands. Second, we slightly change the data placement across DRAM chips in a DIMM to ensure that in-memory aggregation can be performed locally inside each DRAM chip. Such modifications are transparent from other system components.

10.2.2 Programming Model

Our architecture exposes aggregation operations to software as host processor instructions. To simplify the hardware design, we ensure that aggregation operands are always aligned in memory (e.g., eight-byte aggregation operands are aligned to eight-byte boundaries), just as modern compilers guarantee aligned memory access by default. By simply using the new instructions, existing software can benefit from the in-memory aggregation capability. Also, our cache-conscious aggregation automatically adapts to data locality with no software hints (Section 10.2.3), thereby letting compilers aggressively emit aggregation instructions without accurate estimation of data locality. This reduces the burden of developing compilers for our architecture (Section 10.3).

In addition, the simplicity of our programming model helps seamless integration of our idea into existing systems. In particular, AIM supports virtual memory for in-memory aggregation without in-memory MMUs since the target address of an aggregation instruction (which is a virtual address) can be translated into a physical address by using the TLB of the host processor, just as normal load/store instructions.

10.2.3 On-Chip Caches

In our architecture, the host processor executes an aggregation instruction by issuing an aggregation operation, which consists of an aggregation type (e.g., $+$, $\times$, etc.) and an aggregation operand, to the L1 cache. The following describes our cache architecture that supports aggregation operation execution.

Organization. In order to distinguish cache blocks that contain aggregation operands from ordinary blocks, the tag of each block is extended with an aggregation type (AT) field. If the AT field of a block is set to a nonzero value t, the block is called *aggregated block* and should contain *only* the aggregation operands for the aggregation operation of type t. In other words, a single cache block is not allowed to contain aggregation operands from multiple types of aggregation operations. The AT field is set to zero for ordinary cache blocks.

L1 Cache Miss. When an aggregation operation misses in the L1 cache, the L1 cache does *not* read the data from the L2 cache, unlike normal loads/stores. Instead, it simply inserts a new L1 cache block filled with *identity elements*[2] of the given aggregation type t and sets the AT field of the block to t. Since applying an aggregation operation whose operand is an identity element does not change the target data at all, when an aggregated block is merged back to the original data in memory (explained later in this subsection), the portions that are not updated by the host processor are left intact.

However, under the inclusive cache hierarchy, installing a new L1 cache block without accessing the L2 cache violates the inclusion property if the L2 cache does not have the block. Thus, for inclusive caches, the L1 cache sends a special L2 cache request, which lets the L2 cache install an aggregated block filled with identity elements only if the L2 cache does not have the block. This is almost the same as miss handling in conventional cache architectures, except that the contents of the L2 cache block are not transferred into the L1 cache.

L1 Cache Hit. When an aggregation operation hits in the L1 cache, the cache reads the target block, computes the result of the aggregation operation by using its ACU (see Figure 10.3), and stores the result back to the target block, all with no preemption. If the block is an aggregated one, this coalesces the current aggregation operand with the one in the cache, so that they can be sent as a single in-memory aggregation operation on the eviction of the block (explained later). If the block is an ordinary one, the cache directly performs the aggregation operation with the original data.

L1 Cache Block Upgrade. Since aggregated blocks do not store the original data, an aggregated block cannot directly service a normal load/store or an aggregation operation with a different type (commutativity and associativity of aggregation operations hold only for the same type). In such cases, the aggregated block B of type '$*$' is *upgraded*

[2]In mathematics, the identity element of a binary operation '$*$' is defined as x that satisfies $a * x = a$ for all a (e.g., '0' for addition).

to an ordinary one by (1) requesting a normal L2 cache read to fetch its original data D, (2) performing $B *\leftarrow D$ to merge the cache-resident aggregation operands with the original data, and (3) setting the AT field of B to zero (i.e., ordinary).[3] At this point, all cache accesses can be serviced as in other ordinary blocks.

L1 Cache Writeback. When an aggregated block is evicted from the L1 cache, the block is written back to the L2 cache by sending an aggregation operation (instead of a normal write operation) whose type and operand are the value of the AT field and the block data itself, respectively.

Sometimes, a cache block is requested to be written back without being evicted from the cache (e.g., coherence requests, eager writeback, etc.). The L1 cache handles this request by (1) sending an aggregation operation to the L2 cache as described previously, (2) *initializing* the contents of the block with identity elements, and (3) changing the block state from dirty to clean. The last two steps are necessary to prevent the block from being aggregated multiple times.

Upper-Level Caches. All upper-level caches (e.g., L2/L3 caches) operate in the same way as the L1 cache, except for the following differences for the last-level cache. First, when an aggregated block is evicted from the last-level cache, an in-memory aggregation operation is sent to the memory controller. Second, the last-level cache does not send inclusion management requests on misses for an obvious reason.

Example. Figure 10.4 exemplifies an operating sequence of our cache architecture for one cache block, which is initially stored in DRAM and is loaded into two-level inclusive caches. Each set of boxes shows the contents of the block in the L1/L2 cache and the DRAM. For brevity, we assume that the block is two bytes long and each instruction updates a one-byte value (`x` or `y`). White/gray boxes indicate ordinary/aggregated blocks.

At the beginning, the original data `12 34` is stored in the DRAM (a). When the host processor issues $x +\leftarrow 1$, the L1 cache sends an L2 cache request that inserts a new L2 cache block filled with zeros (identity elements of addition) for inclusion management. Then, it initializes a new zero-filled L1 cache block *without accessing the DRAM* and adds one to `x` in the block (b). The subsequent addition to `x` is handled simply by adding its operand to `x` in the L1 cache block (c). When the aggregated L1 cache block is evicted, it is coalesced into the corresponding L2 cache block (d). An addition operation to `y` is

[3] According to our evaluations, such upgrades are infrequent and thus have a negligible energy overhead, i.e., only 2% of the aggregated L3 cache blocks are eventually upgraded to ordinary ones in our large-input workloads.

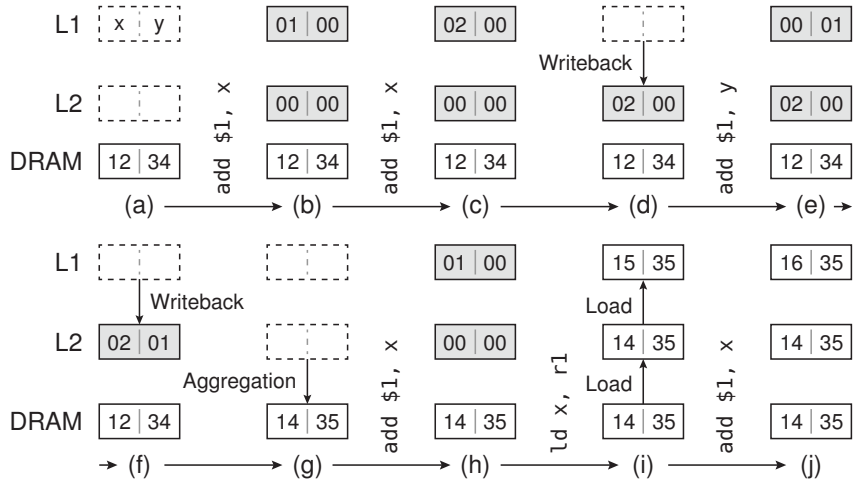


Figure 10.4: An example of cache operations.

performed in the same manner (e). Note that the aggregation operands of both x and y can be stored in a single cache block because their aggregation types are identical (f). The aggregated L2 cache block is later merged into the original data by using in-memory aggregation (g). When a normal instruction accesses an aggregated block, the cache upgrades the block into an ordinary one by loading the original data (h and i). An aggregation operation for an ordinary block is handled in the same way as aggregated blocks (j).

Advantages. Our cache-conscious aggregation provides two major benefits. First, when the target applications have high data locality, our scheme reduces main memory accesses by coalescing multiple aggregation operations to the same data into a single in-memory aggregation operation. Since our mechanism directly uses on-chip caches as locality filters, incorporating any intelligent cache management technique will improve the effectiveness of cache-conscious aggregation.

Second, performing aggregation in caches automatically enables cache coherence support, not only among different cores in the system but also between the host processor and main memory. In a naive approach where the host processor directly sends an aggregation operation to main memory, the memory controller has to make sure that the target cache block is not stored in on-chip caches during the execution of the operation to prevent the host processor from accessing any stale copy of the data. Our architecture does not have such an issue because aggregation operations update on-chip caches first, which makes cache-resident data always up-to-date.

10.2.4 Coherence and Consistency

Since aggregated blocks are the same as ordinary blocks except for their contents, existing cache coherence mechanisms can manage coherence of aggregated blocks with no modifications. This can be shown in the following two steps:

First, if a specific cache block is accessed only by aggregation operations, any cache coherence protocol can maintain the coherence of the block. This is because aggregated blocks implement all types of cache operations supported by ordinary blocks (e.g., hit/miss, writeback, invalidation, etc.) with the same semantics, which allows the coherence protocol to treat aggregated blocks just as ordinary blocks.

Second, even if aggregation operations interact with normal loads/stores, it can still be supported by conventional cache coherence protocols without modifications. This is because such an interaction happens *inside a local cache* after the unmodified cache coherence protocol handles the coherence of the target cache block. For example, if a local core issues a normal load to fetch a cache block that is stored as an aggregated dirty block in a remote cache, our architecture handles this case in two orthogonal steps: (1) flushing the remote block and bringing it into the local cache *as if the local core accesses the block with an aggregation operation* and then (2) locally upgrading the block to an ordinary one. This requires no changes to the cache coherence protocol because (1) is covered by conventional coherence protocols (explained in the previous paragraph) and (2) does not involve coherence protocols at all.

Similarly, aggregation operations are fully compatible with existing consistency models because they are simply treated as a special type of memory write operations. The use of *unmodified* coherence and consistency mechanisms facilitates low-cost, low-effort integration of AIM (e.g., no need for costly in-memory hardware for cache coherence).

10.2.5 Main Memory

DRAM. We add one ACU per DRAM bank to implement in-memory aggregation. Since a bank is the smallest unit of memory-level parallelism in the DDR3 standard, this simplifies resource allocation between aggregation operations and in-memory ACUs. When the memory controller issues an aggregation command to the target DRAM bank, the bank reads the original data from the row buffer, computes the result of the aggregation command, and writes the result.

In addition, implementing in-memory aggregation based on standard memory modules requires modifications to data placement across DRAM chips in a DIMM.

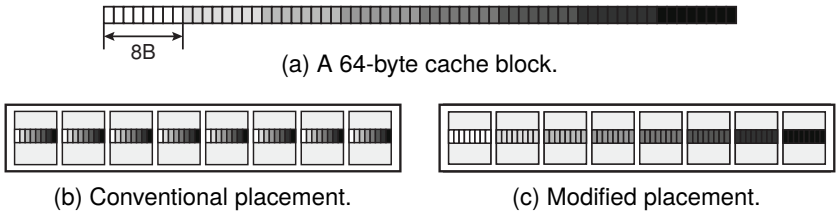


Figure 10.5: Data placement in a DIMM.

Without loss of generality, let us explain this issue with a 64-byte cache block stored in a DDR3 DIMM composed of eight x8 chips. In conventional architectures, each eight-byte subblock is interleaved across eight chips in a one-byte granularity (Figure 10.5b). Under this data placement, performing aggregation for data larger than one byte requires chip-to-chip transfer (e.g., in-memory addition for chip-interleaved 8-byte data requires carry-bit propagation across chips), which is not supported by standard DIMMs. To avoid this, AIM reorganizes the data placement so that each eight-byte subblock is stored inside a single chip (Figure 10.5c). This can be done entirely by memory controllers, and thus, is transparent from other components of the system.

Memory Controllers. Memory controllers schedule aggregation commands in the same way as write commands. This is because the data bus direction of aggregation commands is identical to that of write commands (from memory controllers to DRAM), which is related to the latency of switching the bus polarity in the DDR3 standard (e.g., tWTR).

Depending on the scheduling policy, memory controllers may need to perform aggregation by themselves. This happens when the scheduler reorders a read command to a specific cache block *ahead of previous aggregation commands to the same block*. In that case, after the target block is read from the DRAM, all write/aggregation commands arrived before the read command should be applied to the loaded block to correctly handle data dependence. In reality, such a case occurs rarely since a cache block written back from the last-level cache has low chances to be read again in the near future.

Advantages. Performing aggregation in main memory gives three key advantages in terms of energy efficiency. First, it reduces the maximum number of main memory accesses per aggregation from two (*read*-modify-*write* in conventional architectures) to one (*aggregation*). This is true even for applications with high data locality since in-memory aggregation allows our cache-conscious aggregation to avoid loading the target data from main memory.

Second, it improves DRAM row buffer locality, which is a critical factor in energy efficiency of DRAM. In conventional systems, one aggregation operation usually incurs two row activations because writeback caches delay the write of the aggregation operation until the cache block eviction, which degrades temporal locality between the read and the write of the aggregation operation. On the other hand, AIM implements in-memory aggregation as a single DRAM command, thereby guaranteeing at most one row activation per aggregation.

Third, it reduces switching activities of off-chip memory channels, which leads to lower off-chip I/O energy consumption. This is because aggregation operands tend to have smaller values (i.e., lower entropy in bits) than the original data.

Implications. Supporting in-memory aggregation has implications for some DRAM features. For example, it does not support ECC DIMMs, similar to other PIM architectures based on standard memory modules [60, 176]. Also, the maximum size of an aggregation operand is limited by the DIMM organization (e.g., up to 4 bytes with x4 DRAM, 8 bytes with x8 DRAM, etc.).

However, it should be noted that such implementation issues are due to the underlying main memory technology (i.e., DDR3) rather than our host architecture design (which is our main contribution). Thus, using AIM with other main memory technologies/ organizations (e.g., 3D-stacked DRAM, in-DRAM ECC, etc.) can alleviate such issues. For example, the Hybrid Memory Cube 2.0 standard [28] includes in-memory addition and bitwise commands, which can be seen as a concrete implementation of in-memory aggregation without sacrificing the error correction capability. In our future work, we will explore the impact of other memory technologies on AIM.

10.2.6 Potential Generalization Opportunities

Although we designed AIM for commutative and associative read-modify-write operations (i.e., aggregation), slight modifications to our architecture can potentially support other types of read-modify-write operations as well. Let us assume that there is a pair of operations '$*$' and '$\star$' that satisfies the following property for any a, b, and c (e.g., the below is satisfied if '$*$' is division and '$\star$' is multiplication):

$$a * b * c = a * (b \star c) \tag{10.1}$$

Then, even if '$*$' does not satisfy commutativity and/or associativity, '$*$' can be implemented as aggregation in our architecture by using '$\star$' to merge two aggregation operands. More precisely, the following summarizes how aggregation that supports the above property can be implemented in cache-conscious aggregation:

- When an aggregation operation is performed on an aggregated block, the result is calculated by using '$\star$' operation (instead of '$*$').
- When an aggregation operation is performed on an ordinary block or an aggregated block is upgraded to an ordinary block, the result is calculated by using '$*$' operation.
- When an aggregated block is evicted, an aggregation operation of type '$*$' is sent to the next level of memory.

For example, when $a = a \div b \div c$ is performed, if a is stored in main memory, we can update a by (1) *multiplying* b and c using cache-conscious aggregation and (2) dividing a by $b \times c$ using in-memory aggregation. Although our evaluations do not adopt such extension because our target workloads do not benefit from it (see Section 10.3 for the list of aggregation operations used in our workloads), this could be useful for broadening the applicability of our architecture to applications that extensively use non-commutative and/or non-associative read-modify-write operations.

10.3 Compiler Support

As described in Section 10.2.2, our architecture and its programming model simplify the compiler development for our system. In this section, we demonstrate this advantage by actually developing a compiler for our architecture based on the LLVM compilation framework [177]. We focus on supporting the following four types of aggregation instructions as they are sufficient to cover our target applications (see Section 10.5), but other types of instructions can also be implemented as long as they satisfy the definition of aggregation operations.

- `iadd64`: 64-bit integer add
- `imin64`: 64-bit integer min
- `fadd32`: single-precision floating-point add
- `fadd64`: double-precision floating-point add

Our compiler finds a set of instructions whose semantics match one of the aggregation instructions and *unconditionally* replaces it with the corresponding aggregation instruction. For example, if a `select` instruction (from Figure 10.6) satisfies the following criteria, the compiler always replaces the `select` and its associated instructions (mentioned below) with `imin64`.

```
1    %1 = load i64* %a, align 8
2    %2 = icmp gt i64 %1, %b
3    %3 = select i1 %2, i64 %b, i64 %1
4    store i64 %3, i64* %a, align 8
```

Figure 10.6: Example LLVM language code for `imin64`.

1. One of the source operands (`%1`) of the `select` instruction is from a `load` instruction (line 1).
2. The definition of the `select` instruction (`%3`) is used as a value to be stored in a `store` instruction (line 4).
3. The `load` and the `store` instructions trivially have the same target address aligned to 8 bytes (`%a`).
4. The condition variable (`%2`) of the `select` instruction is from an `icmp` instruction (line 2) that has the same source operands (`%1` and `%b`) as those of the `select` instruction.
5. The condition code (`gt`) of the `icmp` instruction is set in a way that the `select` instruction chooses the smaller value between the two source operands.
6. All uses of intermediate definitions (`%1`, `%2`, and `%3`) are restricted to the instructions mentioned above.
7. All instructions belong to the same basic block and there are no other memory write instructions between the `load` and the `store` instructions.

Through our evaluations, we will show that this simple approach *without compile-time data locality estimation* (which could be tricky or even impractical) works nicely due to our cache-conscious aggregation, thereby simplifying the toolchain development for PIM systems.

10.4 Contributions over Prior Art

In this section, we summarize the contributions of this work over the state of the art.

10.4.1 PIM-Enabled Instructions

Programming Model. Compared to PEI, AIM provides a much more intuitive programming model, which enables *zero modifications to existing software*. First,

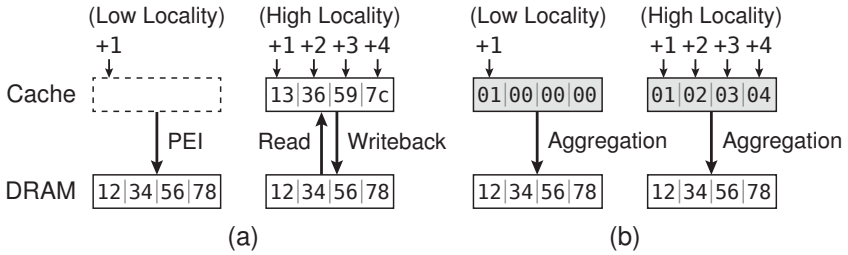


Figure 10.7: Exploiting data locality in (a) PEI and (b) AIM.

contrary to PEI which needs explicit synchronization between PIM instructions and normal instructions by issuing pfence, cache-conscious aggregation of AIM eliminates such a requirement. Second, PEI exposes separate operand storage (called operand buffer) to software, whereas AIM directly uses host processor registers as operands of aggregation instructions. Consequently, our aggregation instructions serve as *drop-in* replacements for the equivalent normal instructions, which simplifies the compiler design (e.g., no need for compilers to identify the places to insert pfence). Although AIM does not currently support non-aggregation operations, it can be used together with PEI to make AIM-compatible instructions more energy-efficient while supporting other PIM-enabled instructions as well.

In addition, AIM provides a fully automated compilation flow, whereas PEI is evaluated with hand-written code only.

Locality-Aware Execution. While PEI offloads PIM operations to either the host processor or main memory, AIM is able to execute aggregation operations at any level of the memory hierarchy, thereby enabling better adaptation to data locality. For instance, if multiple aggregation operations access the same cache block (see Figure 10.7), AIM accumulates their operands inside the block and sends it to main memory with in-memory aggregation, whereas PEI lets the host processor execute all of them after fetching the block from main memory and then writes the result back to main memory (thereby generating more main memory accesses than AIM).

Moreover, PEI has higher chances to mispredict data locality than AIM due to its separate locality monitor structure. For example, since the locality monitor can check locality only *after* a PEI is executed, the first PEI to each cache block is always offloaded to main memory regardless of the locality of the block. Note that, unlike our cache-conscious aggregation, the host-/memory-side PEI execution mechanism inevitably requires a separate structure for locality monitoring.

Coherence Support. Unlike PEI which needs to send back-invalidation (or back-writeback) to the last-level cache for coherence management, AIM supports cache coherence without such requests as cache-resident data is always up to date.

Main Memory. While PEI is evaluated based on HMCs, AIM uses the widespread DDR3 memory modules by default. Although both PEI and AIM do not rely on a particular memory technology, implementing a locality-aware PIM system with DDR3 main memory is more challenging because the lack of support for fine-grained (e.g., 8-byte) memory accesses increases the memory bandwidth cost of locality misprediction (e.g., using PIM for high-locality data). This requires more accurate locality adaptation, which is why AIM is necessary.

Energy Efficiency. Our work finds that processing aggregation inside the memory hierarchy reduces row activations, main memory accesses, and off-chip channel bit-flips, thereby improving the energy efficiency of the memory hierarchy. This has not been evaluated before.

10.4.2 Parallel Reduction in Caches

The concept of merging aggregation in caches has also been used to improve the scalability of parallel reduction. Although such a benefit is completely orthogonal to our work, we compare AIM against two parallel in-cache reduction mechanisms: PCLR [178] and Coup [179]. Both techniques alleviate communication overheads of reduction by letting each core locally perform reduction on its private cache blocks, but unlike our work, they do not explore the energy perspective of reduction.

Energy Efficiency. Since the purpose of PCLR and Coup is to reduce the cost of parallel updates in multiprocessor systems, both of them merge reduction operands with their original data *in the closest shared memory to the processors*, that is, shared caches in modern computer systems. Thus, when the target cache block of reduction is not present in shared caches, they have to load the original data from main memory as in conventional architectures. Considering that our in-memory aggregation reduces main memory accesses, row activations, and off-chip channel bit-flips by *avoiding loading the original data of aggregation from main memory*, PCLR and Coup cannot improve the energy efficiency of main memory unlike AIM (see Section 10.7.1 for a quantitative evaluation of this aspect). Note that extending PCLR or Coup to support in-memory aggregation is not trivial as the shared cache should be able to distinguish reduction

data from normal data and propagate reduction operands from shared caches to main memory, which requires our cache-conscious aggregation.

Applicability. PCLR and Coup are effective only if (1) multiple cores competitively access the reduction data and (2) the reduction data fits in the shared last-level cache (since they have to fetch the original data on shared cache misses). On the contrary, AIM becomes more energy-efficient with a larger volume of data and does not depend on whether the data is shared or not. Hence, AIM has broader use cases than PCLR and Coup, considering that the amount of data handled by emerging data-intensive workloads is already far beyond the last-level cache capacity and is still explosively increasing.

Effort for Integration. Compared to AIM, PCLR and Coup require more effort to be integrated into existing systems. First, PCLR is not general enough to be used beyond parallel reduction (e.g., unable to share data between reduction and normal instructions, manual cache block flushes required at the end of reduction loops, etc.). Second, Coup introduces significant changes to the cache coherence protocol, which increases the design cost of verifying the coherence protocol and, particularly, its implementation. Third, PCLR and Coup assume manual modification of existing software, whereas AIM eliminates such an obstacle by providing a fully automated compiler.

10.4.3 Row Buffer Locality of DRAM Writes

Since conventional writeback caches degrade the row buffer hit ratio of DRAM writes, there have been many techniques to improve the row buffer locality of DRAM writes by issuing last-level cache writebacks earlier than cache block evictions [109,130,180–182]. Compared to such techniques, AIM offers the following two benefits. First, in-memory aggregation *guarantees* the write of an aggregation operation to hit in the row buffer. Second, AIM does not generate extra writebacks since it *delays the read* of an aggregation operation until the corresponding write, instead of proactively issuing extra writebacks. We will quantitatively analyze these two aspects in Section 10.7.3.

10.5 Target Applications

In this section, we describe our target multithreaded workloads from both standard benchmarks and emerging data-intensive applications. These workloads are important in that they are often performance bottlenecks of many important applications (e.g., graph algorithms in social network analysis, backpropagation in deep learning, sparse

matrix-vector multiplication in scientific computing, etc.). All workloads are compiled by our compiler with no software modifications.

Average Teenage Follower (AT) [149] is an example kernel of social network analysis. For each teenager vertex, the follower counts of its successors are incremented by using `iadd64`, which calculates the number of teenage followers.

Backpropagation (BP) [183] is a widely-used algorithm for training neural networks. AIM uses `fadd32` to subtract a ratio of gradients from the weights (i.e., backward propagation of the error between the prediction and the expected outcome).

Breadth-First Search (BF) [149] is a graph traversal algorithm. We evaluate the level-synchronous BFS [149, 166], in which each vertex is equipped with a 'level' field to indicate the breadth of the vertex. For each iteration, vertices in the current level update the levels of their successors with the min function (`imin64`) to ignore already-visited vertices.

Histogram (HG) [184] calculates the distribution of data by counting items that are mapped to each bin. AIM increments the bin counter for each data item by using `iadd64`.

HotSpot (HS) [183] performs 2D transient thermal simulation of VLSI systems. For each iteration, the temperature of each grid is adjusted by estimating the effect of adjacent grids. For this computation, AIM uses `fadd64` to add the contribution from nearby grids to the temperature of the current grid.

PageRank (PR) [2, 137, 149, 151] computes the importance of each vertex in a graph from the relationship between vertices. AIM uses `fadd64` for updating the rank of successor vertices.

RabbitCT (RC) [185] provides a multicore implementation of the FDK algorithm [186], which performs backprojection for CT (computed tomography) image reconstruction. In this workload, AIM uses `fadd32` for overlaying 2D measurements into the corresponding 3D space to reconstruct a 3D image.

Sparse matrix-vector multiplication (SM) [187] multiplies a sparse matrix A and a vector x, which is an important kernel of many scientific applications. It uses the compressed sparse column (CSC) format, which generates random memory additions (`fadd64` in AIM) to the output vector.

Single-Source Shortest Path (SP) [2, 149] is the parallel Bellman-Ford algorithm for finding the shortest paths from a source vertex to all other vertices in a graph. In this algorithm, AIM uses `imin64` for edge relaxation, in which a vertex u updates the distance d_v of its successor v to $\min(d_v, d_u + w_{uv})$.

Table 10.1: Baseline Simulation Configuration

Component	Configuration
Core	16 out-of-order cores, 4 GHz, 4-issue, 128-entry instruction window
L1 I/D-Cache	Private, 32 KB, 4/8-way, 64 B blocks, 16 MSHRs
L2 Cache	Private, 256 KB, 8-way, 64 B blocks, 16 MSHRs
L3 Cache	Shared, 16 MB, 16-way, 64 B blocks, 64 MSHRs
DRAM Controller	32-entry read/write queue, FR-FCFS scheduling, drain-when-full write buffer policy [180]
DRAM	16 GB, DDR3-1600 11-11-11 [96], 4 channels, 1 rank/channel, 8 banks/rank, 8 KB rows

Weakly-Connected Components (WC) [167] is the HCC algorithm [167] for finding weakly connected components in a graph. It initializes each vertex with a unique integer label and, for each iteration, collapses the labels of adjacent vertices into the smallest one among them by using `imin64`.

10.6 Evaluation Methodology

10.6.1 Simulation Configuration

We evaluate our architecture based on our x86-64 simulator whose frontend is Pin [95]. Our simulator has cycle-level timing models of out-of-order cores considering register/structural dependency and limited instruction window, three-level inclusive multi-bank caches with MSHRs, the MESI cache coherence protocol, on-chip crossbar networks, multi-channel memory controllers, and DDR3-based main memory. The simulation configuration of the baseline system is summarized in Table 10.1. We assume that the host processor of the baseline also implements normal instruction versions of `iadd64`, `imin64`, `fadd32`, and `fadd64` so that all systems execute the same number of instructions for the same amount of work.

For energy evaluation, we use Micron's DDR3 SDRAM System Power Calculator [96, 100] to estimate the energy consumption of DRAM and utilize CACTI-3DD [126] to further break down the energy inside a DRAM chip, similar to previous work [188]. Also, the energy consumption of on-chip caches and memory controllers is modeled by CACTI 6.5 [13] and McPAT 1.2 [127], respectively.

For in-memory aggregation, each DRAM bank incorporates one 64-bit ACU, which includes one 64-bit integer adder (`iadd64`), one 64-bit integer comparator/multiplexer

(`imin64`), two single-precision floating-point adders (`fadd32`), and one double-precision floating-point adder (`fadd64`). In-memory ACUs are assumed to have 5 ns of delay, or one-cycle delay at the DRAM core clock, with no pipelining.

Also, on-chip caches are equipped with 8-way 64-bit ACUs to compute the result of aggregation for a 64-byte cache block. In our configuration, each L1 cache bank has one ACU, while eight/four L2/L3 cache banks share one ACU.[4] We design the on-chip ACU to have one-/five-cycle delay at 4 GHz with no pipelining for an integer/floating-point operation.

10.6.2 Hardware Overhead

The following explains the energy/area overheads of hardware modifications introduced by AIM and our methodology to obtain such estimates. The experimental results shown in this chapter take these energy overheads into account.

In-Memory ACUs. To estimate the energy/area overheads of in-memory ACUs, we synthesize our Verilog implementation of the ACU design by using Synopsys Design Compiler. Since proprietary DRAM processes are not available due to confidentiality, we conduct conservative estimation by using the TSMC 130 nm technology, considering that DRAM processes fall behind logic processes by multiple generations in implementing logic circuits [58]. According to the synthesis result, eight 64-bit ACUs (one per bank) take only 0.42 mm^2 of die area, which is negligible compared to large modern DRAM die sizes (e.g., 30.9 mm^2 for a 23 nm 4 Gb DRAM die [189]). Due to this wide area gap between ACUs and DRAM dies, although the exact amount of discrepancy between DRAM processes and logic processes can vary, ACUs implemented by DRAM processes are still expected to have a small area overhead. If such extra cost is unaffordable, in-memory ACUs can also be implemented on the existing logic die of 3D-stacked DRAM (e.g., in-memory atomics of HMCs [28]) as our mechanism is not limited to a particular main memory technology.

On-Chip ACUs. To estimate the energy/area overheads of on-chip ACUs, we synthesize the ACU design with our timing constraint by using the TSMC 45 nm technology library and compare the result against the cache area modeled by CACTI. The area overhead of on-chip ACUs is 3% of the on-chip cache hierarchy area.

[4] We performed a parameter sweep of the number of ACUs per cache and found that putting one dedicated ACU to every L2/L3 cache bank shows less than 1% energy reduction compared to our shared ACU configuration.

Table 10.2: Input Sets of Workloads

Application	Small Input	Large Input
AT, BF, PR, SP, and WC	soc-Slashdot0811 [162]	soc-LiveJournal1 [162]
BP	$65536 \times 16 \times 1$ [183]	$4096 \times 4096 \times 1000$ [190]
HG	1024 bins	4,194,304 bins
HS	256×256 grids	1024×1024 grids
RC	256^3 voxels [185]	1024^3 voxels [185]
SM	AMD/G2_circuit [191]	Freescale/FullChip [191]

Aggregation Type Fields in Cache Tags. AIM adds only three extra bits per tag across all caches to store one of the five possible aggregation types (i.e., ordinary, `iadd64`, `imin64`, `fadd32`, and `fadd64`). This introduces negligible area overheads to on-chip caches (e.g., less than 1% of storage overhead).

10.6.3 Workloads

We evaluate ten data-intensive applications explained in Section 10.5. Each workload has two input sets with different sizes (shown in Table 10.2) to demonstrate the impact of the working set size. All workloads are simulated for up to one billion instructions after skipping initialization phases. In these workloads, aggregation instructions account for 4% of the dynamic instruction count on average; however, they are responsible for 50% of the total DRAM accesses in large-input workloads. Since our target applications are mostly memory-bandwidth-bound, efficient aggregation execution is very important in our workloads.

10.7 Evaluation Results

10.7.1 Energy Consumption and Performance

Figure 10.8 compares the energy consumption and system performance of the following five configurations. All results are normalized to the baseline and the rightmost sets of bars labeled as 'GM' indicate the geometric mean of the results.

- *Baseline* is the traditional system shown in Table 10.1.
- *PIM-Only* executes all aggregation operations in main memory after flushing the target block from on-chip caches (no cache-conscious aggregation).

- *PEI* represents PIM-enabled instructions [155] (see Section 10.4.1). Contrary to the original paper whose evaluation is based on HMC, we use the main memory architecture of AIM (based on DDR3) as in-memory PIM operation implementation for a fair comparison.
- *AIM-Private* enables cache-conscious aggregation only for private caches (i.e., L1/L2 caches). Thus, when an aggregation operation causes an L3 cache miss, the L3 cache always fetches the target data from main memory.
- *AIM* is the proposed architecture supporting both cache-conscious aggregation and in-memory aggregation.

AIM vs. Baseline/PIM-Only. From Figure 10.8, we draw three conclusions. First, AIM reduces the energy consumption of main memory by 15%/28% in small-/large-input workloads (see Section 10.7.2 for detailed analysis). Note that simply using in-memory aggregation without proper consideration of data locality (i.e., PIM-Only) *increases* the average main memory energy consumption by 5.4x in small-input workloads. This is because PIM-Only does not utilize on-chip caches for aggregation operations, and thus, generates 8.8x as many main memory accesses as the baseline does in small-input workloads. The same holds even for some large-input workloads with good data locality (e.g., HS, RC, and SM).

Second, AIM consumes 13%/19% less on-chip memory hierarchy energy in small-/large-input workloads, respectively. This comes mostly from the performance improvement of AIM (see the next paragraph). Although our cache-conscious aggregation may perform more computation than the baseline (e.g., upgrades from aggregated blocks to ordinary ones), its impact is negligible as discussed in footnote 3.

Third, AIM also improves average system performance by 21%/29% for small-/large-input workloads due to two major factors. First, fewer main memory accesses in AIM alleviate the off-chip memory bandwidth bottleneck. Second, AIM can move main memory accesses for aggregation operations out of the critical path of program execution. In conventional architectures, when the target data of aggregation is stored in main memory, the host processor has to wait until the data is loaded from the main memory. On the other hand, AIM simply initializes the L1 cache block with identity elements and executes the aggregation operation on it. This shortens the processor stall time caused by aggregation operations.

AIM vs. PEI. We also compare AIM with PEI [155]. In small-input workloads, AIM achieves 19% lower main memory energy consumption and 31% higher performance than

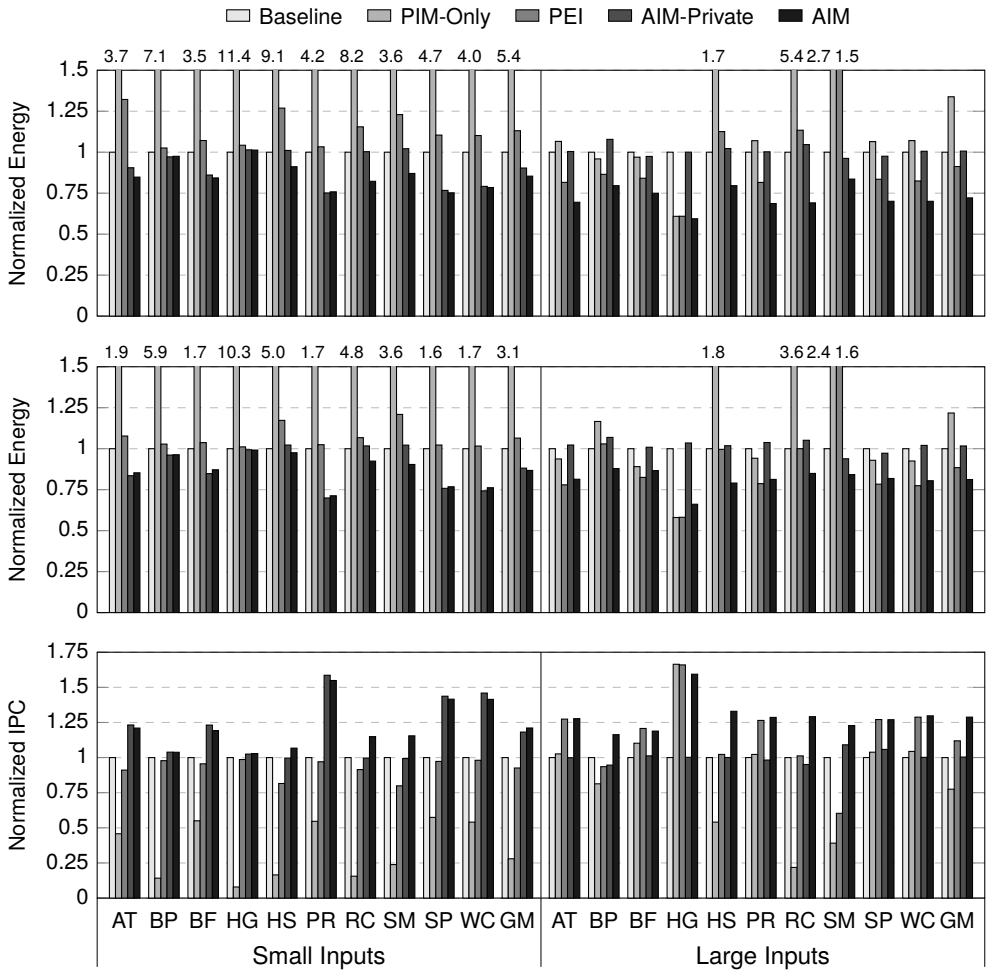


Figure 10.8: Comparison of the energy consumption of DRAM (top), the energy consumption of on-chip caches and memory controllers (middle), and the system performance (bottom).

PEI. This is because (1) AIM is able to utilize PIM even under high data locality unlike PEI and (2) the locality prediction of PEI is less accurate than AIM (see Section 10.4.1). Although the original paper [155] showed that PEI matches the performance and energy consumption of host-side execution in small-input workloads, this does not happen under our configuration since our DDR3-based system imposes higher memory bandwidth overheads to locality misprediction than the original HMC-based system (discussed in Section 10.4.1).

In large-input workloads, AIM reduces the main memory energy consumption by 21% and improves the performance by 15% compared to PEI. Ideally, PEI and AIM should perform almost identically under zero data locality as both will always utilize PIM. However, in reality, even large-input workloads have some degree of locality, which makes AIM outperform PEI as in small-input workloads. This is especially noticeable in HG, RC, and SM because they exhibit higher data locality (i.e., 14% higher average cache hit ratio in the baseline) than the rest of the large-input workloads.

AIM vs. AIM-Private. Lastly, we evaluate AIM against AIM-Private. Since AIM-Private merges aggregation operands with their original data *in the shared last-level cache* (rather than DRAM as in AIM) just as Coup [179] does, comparing AIM against AIM-Private shows the advantages of our cache-conscious aggregation over parallel in-cache reduction techniques such as PCLR and Coup (see Section 10.4.2). Note that, although Coup improves the scalability of parallel reduction unlike AIM-Private, it is orthogonal to the benefit of AIM.

According to the experimental results, AIM-Private is not effective at all when the working set size exceeds the cache capacity. In large-input workloads, AIM-Private shows almost the same DRAM energy consumption as the baseline, whereas AIM achieves 28% reduction (they show comparable energy efficiency in small-input workloads). This is because AIM-Private issues plain DRAM reads/writes to handle aggregation operations for memory-resident data just as conventional architectures do. From this result, we can conclude that (1) the energy efficiency benefit of parallel in-cache reduction techniques is *not scalable*, in that they are limited to the case when the working set size does not exceed the last-level cache capacity,[5] and (2) our new architectural design overcomes this limitation by exposing aggregation to all levels of the memory hierarchy, which we call a *processing-in-memory-hierarchy* paradigm.

[5]This is an important limitation because the working set size of our target workloads (and many other big-data workloads) easily exceeds the last-level cache capacity.

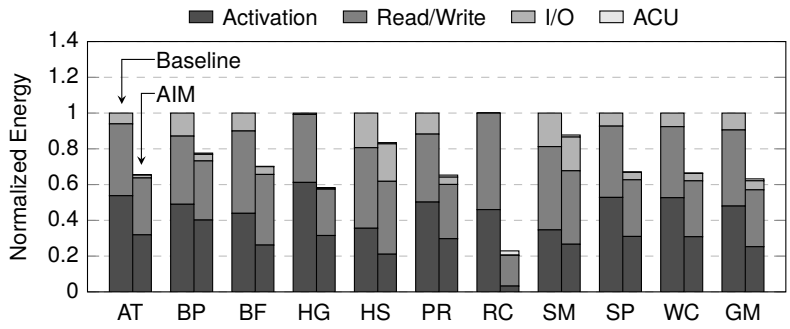


Figure 10.9: DRAM dynamic energy breakdown.

10.7.2 Dynamic Energy Breakdown

Figure 10.9 shows DRAM dynamic energy breakdown in large-input workloads. We omit background energy of DRAM (39% of the DRAM energy in the baseline) as it is simply proportional to execution time. Also, results for small-input workloads are not shown here because, under high data locality, on-chip caches coalesce most of the aggregation operations, and thus, DRAM energy analysis in such cases provides little insight into the benefit of in-memory aggregation.

Most noticeably, AIM reduces the average energy consumption of row activations by 50%. This is contributed by the fact that AIM guarantees at most one row activation per aggregation operation, contrary to the baseline where one aggregation operation incurs up to two row activations. Due to this, AIM achieves a 30% higher average row buffer hit ratio than the baseline, which indicates fewer row activations.

Moreover, AIM saves 50% of the off-chip I/O energy consumption on average due to two reasons. First, it eliminates 36% of main memory accesses by reducing the maximum number of main memory accesses per aggregation operation from two (read and write) to one. Second, it reduces off-chip channel bit-flips per transfer by 48% since aggregation operands usually have smaller values than the original data.

Lastly, AIM reduces the read/write energy consumption by 26% on average because one aggregation command consumes lower energy than a combination of one read command followed by one write command (i.e., 33% lower energy according to our model based on CACTI-3DD [126]). The reason for this is that (1) the data read by an aggregation command does not need to be transferred outside the bank and (2) in-memory aggregation sends one less command to the bank.

All these benefits are achieved with very low implementation overheads. On average, in-memory ACUs contribute only 1.7% of the DRAM energy consumption.

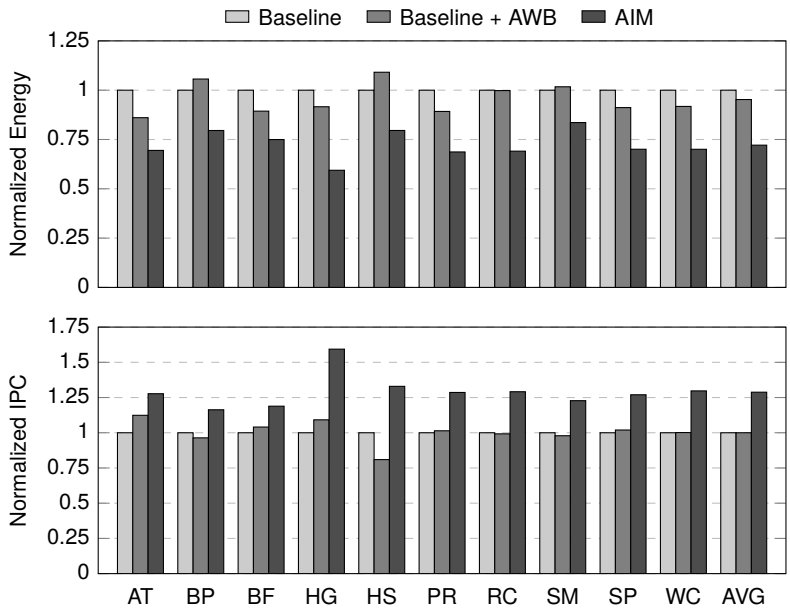


Figure 10.10: Effect of aggressive writeback on DRAM energy (top) and system performance (bottom).

10.7.3 Comparison with Aggressive Writeback

As explained in Section 10.4.3, there have been several techniques that improve the row buffer locality of conventional architectures by proactively issuing bulk writebacks to cache blocks in the same DRAM row. However, as shown in Figure 10.10, even after the baseline adopts aggressive writeback (AWB) [182], AIM provides higher energy efficiency than the baseline. This is because, although AWB does shorten the distance between the read and the write of each aggregation operation to some extent, it cannot schedule *every* write of an aggregation operation *right after* the corresponding read. On the other hand, AIM *guarantees* row buffer hits on the writes of aggregation operations, thereby achieving a 13% higher row buffer hit ratio than the baseline with AWB.

Also, AWB sometimes degrades system performance due to two reasons. First, proactively issuing writebacks incurs up to 19% of extra DRAM writes, which is harmful to applications with high memory bandwidth consumption. Second, extra writeback requests increase last-level cache contention, which may block latency-critical reads/writes from the host processor. AIM is free from both drawbacks (e.g., up to 2% increase in DRAM writes), and thus, it does not degrade the performance across all applications evaluated in this chapter.

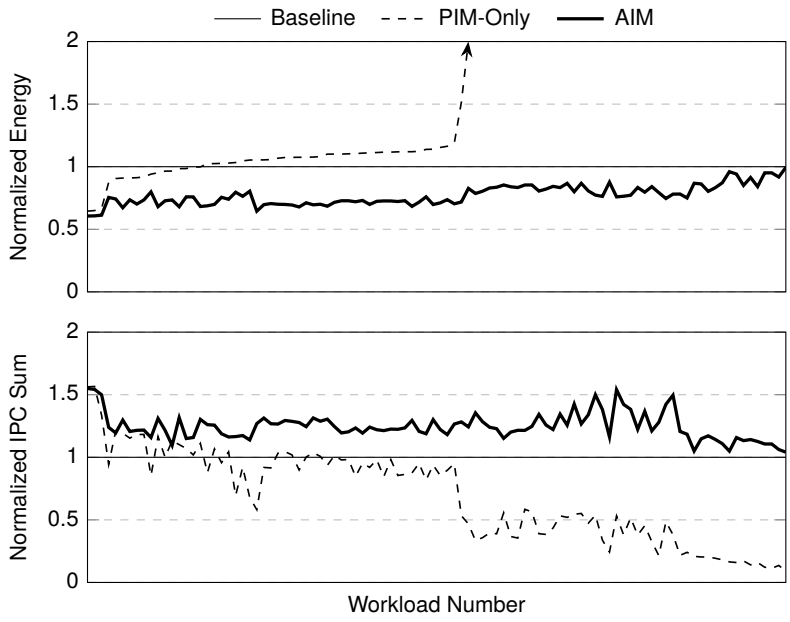


Figure 10.11: DRAM energy consumption (top) and performance (bottom) in multiprogrammed workloads.

10.7.4 Multiprogrammed Workloads

In order to show the robustness of our cache-conscious aggregation against varying data locality, we evaluate our architecture by using 100 multiprogrammed workloads. Each workload consists of two applications that are randomly selected from our target applications. All results are sorted by the energy consumption of PIM-Only. We omit the normalized energy consumption higher than 2.0 for better visibility.

As shown in Figure 10.11, AIM consistently outperforms *both* the baseline and PIM-Only in terms of energy efficiency and performance. The results also confirm that always using in-memory aggregation is harmful to both energy consumption (up to 10x, not shown in the figure) and performance (up to 91% slowdown). We conclude that the adaptivity of AIM is robust enough to be used in real-world situations where a single machine dynamically services multiple workloads with diverse data locality.

10.7.5 Comparison with Intrinsic-based Code

To demonstrate the quality of our compiler, we compare the energy consumption and performance of the compiler-generated binaries (shown in Figure 10.8) with those

of intrinsic-based, hand-written code where aggregation instructions are manually inserted to software. According to our analysis, our compiler-based approach identifies all opportunities to use aggregation instructions in our workloads with no programmer intervention and shows only 0.3% higher DRAM energy consumption and 0.6% lower performance than the intrinsic-based approach on average.

10.8 Summary

We proposed aggregation-in-memory (AIM), a new style of PIM systems designed from the ground up for energy efficiency. AIM features two key contributions that realize processing-in-memory-hierarchy: (1) in-memory aggregation, which tackles the type of computation that is difficult for the traditional memory hierarchy to handle in an energy-efficient way, and (2) cache-conscious aggregation, which leverages on-chip caches for dynamically coalescing aggregation operations with high data locality before they are sent to main memory. These new ideas are seamlessly integrated into existing systems with an intuitive programming model and hassle-free compiler support. Our extensive evaluations show that AIM greatly improves both energy efficiency and system performance by reducing main memory accesses, improving DRAM row buffer locality, and reducing switching activities of memory channels. We conclude that AIM paves the way for introducing the PIM concept into existing systems in the most energy-efficient and least disruptive way possible.

Chapter 11

Conclusion

With the continuous development of architectural techniques that improve the efficiency of computational units and the increasing importance of data-intensive workloads, performance and energy efficiency of modern computer systems are now largely dominated by memory system design. Unfortunately, the efficiency of a traditional memory hierarchy has reached its limitation mainly due to the underlying memory technologies. First, low-density, energy-hungry SRAM hinders realization of very large on-chip caches in a cost-effective, energy-efficient manner. Second, off-package DRAM connected via narrow-bandwidth channels limits how fast data can be retrieved and processed by computational units.

In this dissertation, we showed that such inefficiency of traditional memory hierarchies can be overcome in a practical way by *leveraging emerging memory technologies and re-designing the memory system to be aware of such new memory technologies*. In particular, we focused on two new technologies, STT-RAM for on-chip caches and logic-enabled DRAM for main memory, both of which have actively developed by academia and industry in recent years. Throughout this dissertation, we thoroughly addressed the challenges in designing a memory hierarchy based on such technologies in a way to maximize their advantages while minimizing the impact of their drawbacks and the effort to integrate them into existing systems.

11.1 Energy-Efficient On-Chip Caches based on STT-RAM

In the first part of this dissertation, we proposed four on-chip cache architectures based on STT-RAM. As a motivational result, we showed that STT-RAM caches provide several advantages over traditional SRAM caches, including non-volatility, low static power, and high density, but inefficient write operations of STT-RAM can easily offset such benefits. Our four architectural techniques addressed this challenge in unique ways.

First, we proposed an STT-RAM instruction cache architecture that leverages its non-volatility to reduce the static energy consumption (Chapter 3). We observed that, since write operations are relatively infrequent in instruction caches, the energy consumption of an STT-RAM instruction cache is dominated by its static energy consumption. In order to mitigate this issue, our mechanism exploits the locality of instruction access inside a loop by turning off the STT-RAM instruction cache inside a small loop while instructions are served by a small SRAM cache called loop cache. This reduces the energy consumption of an STT-RAM instruction cache by 49% on average.

Second, we demonstrated that the narrow bit-width characteristics of application data can be utilized to reduce the write energy consumption of STT-RAM data caches (Chapter 4). The key idea of our approach, lower-bits cache, is to filter out frequent lower bit changes from the STT-RAM L2 cache by adding a small SRAM cache that stores only the lower bits of frequently modified data. On average, this reduces 46% of dynamic energy consumption of an STT-RAM L2 cache.

Third, we showed that the write intensity of a cache block can be predicted at the time of its insertion and such information can be very useful in devising an accurate block placement policy for SRAM/STT-RAM hybrid caches (Chapter 5). Our proposal, prediction hybrid caches, includes a new hardware structure called write intensity predictor, which predicts the write intensity of cache blocks based on its correlation with instruction addresses, and allocates predicted write-intensive blocks to the SRAM region in hybrid caches. The proposed mechanism achieves 28% and 31% reductions in L2 cache energy consumption in a single-core (quad-core) system.

Fourth, we identified dead writes, a new type of write operations that can safely bypass the cache without incurring extra cache misses (Chapter 6). Our new hardware architecture, DASCA, accurately predicts all three cases of dead writes (i.e., dead-on-arrival fills, dead-value fills, and closing writes) and lets the last-level cache bypass the predicted dead writes to reduce its write energy consumption. On average, DASCA achieves 68%/62% last-level cache energy reductions, 10%/16% main memory energy reductions, and 6%/14% performance improvements in a single-/quad-core system.

We believe that these four techniques contributed to making STT-RAM attractive as a key enabler technology for very large energy-efficient on-chip caches. Considering that vanilla STT-RAM caches are not competitive against state-of-the-art SRAM caches, we expect that such architectural approaches will play an important role in realizing STT-RAM caches as a promising alternative to conventional SRAM caches.

11.2 Intelligent Main Memory based on Logic-Enabled DRAM

In the second part of this dissertation, we proposed four intelligent main memory systems based on logic-enabled DRAM. Our motivation is that, unlike early PIM proposals based on significant modifications to DRAM dies, logic-enabled DRAM based on 3D stacking realizes practical, low-cost implementation of complex logic circuits inside main memory. With this trend, our four system designs explored the benefits of tight integration of logic and memory while carefully considering the interoperability with existing systems.

First, we showed that the existing logic die of a commercial logic-enabled DRAM product, Hybrid Memory Cube, can be utilized to perform link power management and low-overhead aggressive prefetching (Chapter 7). We observed that high off-chip bandwidth provided by high-speed serial links of HMCs are not always fully utilized, in which case they incur a significant amount of energy overheads. Our mechanism minimizes this overhead by dynamically disabling some of the links according to the actual link loads and reducing off-chip bandwidth consumption from aggressive prefetchers through in-memory prefetch buffer. It reduces the energy consumption of HMCs by 52% on average with minimal performance degradation.

Second, we identified large-scale graph processing as a potential killer application for processing-in-memory and demonstrated its benefit by developing a scalable PIM accelerator architecture for graph processing (Chapter 8). Our accelerator design, Tesseract, employs an efficient communication mechanism to hide long remote access latency in graph processing and specialized prefetchers that utilize the domain-specific knowledge to maximize internal bandwidth utilization. Tesseract achieves a 14x average performance improvement over a conventional DDR3-based system and, more importantly, provides performance proportional to main memory capacity, thereby allowing us to scale the system performance by simply adding more memory modules.

Third, we developed a new style of PIM systems called PIM-enabled instructions, which aims at integrating the PIM concept to existing systems with minimal effort (Chapter 9). We showed that, by extending the existing sequential programming model, PIM can be seamlessly integrated into conventional systems while supporting and

leveraging existing mechanisms that many software applications are already based on, such as cache coherence and virtual memory. Moreover, the instruction-based PIM programming model opened up the new paradigm of locality-aware PIM execution, where the location of PIM operation execution is dynamically determined based on target data locality. Our architecture design that implements PIM-enabled instructions achieves 47% performance improvement over conventional systems in large-input workloads, while maintaining the same level of performance in small-input workloads where a PIM-only approach shows 20% performance degradation.

Fourth, we introduced a more advanced locality-aware PIM architecture designed for energy efficiency, called aggregation-in-memory (Chapter 10). Based on the observation that offloading aggregation operations to memory can greatly improve the energy efficiency of main memory, we developed a host-side architecture that allows aggregation operations to be executed at any level of the memory hierarchy depending on the data locality. Furthermore, we presented a fully automated compiler toolchain for our architecture, which transforms existing applications to use new PIM operations without manual modifications to the software codebase. Our architecture achieves 15%/28% main memory energy reductions and 21%/29% performance improvements over conventional systems in small-/large-input workloads.

We conclude that our four holistic approaches established the foundation of practical intelligent main memory system design from hardware architectures to programming models and compiler support. Our work explored a wide range of PIM system design from maximal performance and scalability to minimal modifications and seamless integration, which is very useful for future PIM research. With the growing interest in data-intensive computing and near-data processing, we anticipate that our work will have long-term impacts on both academia and industry.

Bibliography

[1] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the International Conference on Management of Data*, 2010, pp. 135–146.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[4] Oracle TimesTen in-memory database. [Online]. Available: http://www.oracle.com/technetwork/database/timesten/

[5] SAP HANA. [Online]. Available: http://www.saphana.com/

[6] S. Rusu, H. Muljono, D. Ayers, S. Tam, W. Chen, A. Martin, S. Li, S. Vora, R. Varada, and E. Wang, "Ivytown: A 22nm 15-core enterprise Xeon processor family," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2014, pp. 102–103.

[7] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the International Symposium on Computer Architecture*, 2015, pp. 105–117.

[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[9] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti, C. Houghton, D. Krueger, O. Franza, J. Desai, J. Crop, B. Brock, D. Bradley, C. Bostak, S. Bhimji, and M. Becker, "The Xeon® processor E5-2600 v3: A 22 nm 18-core product family," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 92–104, Jan. 2016.

[10] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for CMP scaling," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 371–382.

[11] A. Driskill-Smith, D. Apalkov, V. Nikitin, X. Tang, S. Watts, D. Lottis, K. Moon, A. Khvalkovskiy, R. Kawakami, X. Luo, A. Ong, E. Chen, and M. Krounbi, "Latest advances and roadmap for in-plane and perpendicular STT-RAM," in *Proceedings of the International Memory Workshop*, 2011, pp. 1–3.

[12] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 9, no. 2, pp. 13:1–13:35, May 2013.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, Apr. 2009.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[15] Y. Zhang, W. Wen, and Y. Chen, "The prospect of STT-RAM scaling from readability perspective," *IEEE Transactions on Magnetics*, vol. 48, no. 11, pp. 3035–3038, Nov. 2012.

[16] P. M. Kogge, "EXECUBE-a new architecture for scaleable MPPs," in *Proceedings of the International Conference on Parallel Processing*, 1994, pp. 77–84.

[17] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995.

[18] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): Chips that remember and compute," in *International Solid-State Circuits Conference Digest of Technical Papers*, 1997, pp. 224–225.

[19] ——, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.

[20] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the International Symposium on Computer Architecture*, 1998, pp. 192–203.

[21] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *Proceedings of the International Conference on Computer Design*, 1999, pp. 192–201.

[22] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proceedings of the Conference on Supercomputing*, 1999, p. 57.

[23] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul.–Aug. 2014.

[24] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proceedings of the Symposium on VLSI Technology*, 2012, pp. 87–88.

[25] *High Bandwidth Memory (HBM) DRAM*, JEDEC Std. JESD235A, Nov. 2015.

[26] "Hybrid memory cube specification 1.0," Hybrid Memory Cube Consortium, Tech. Rep., Jan. 2013.

[27] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 190–200.

[28] "Hybrid memory cube specification 2.0," Hybrid Memory Cube Consortium, Tech. Rep., Nov. 2014.

[29] X. Dong, X. Wu, G. Sun, Y. Xie, H. H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement," in *Proceedings of the Design Automation Conference*, 2008, pp. 554–559.

[30] X. Dong, X. Wu, Y. Xie, Y. Chen, and H. Li, "Stacking magnetic random access memory atop microprocessors: An architecture-level evaluation," *IET Computer & Digital Techniques*, vol. 5, no. 3, pp. 213–220, May 2011.

[31] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009, pp. 239–249.

[32] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 34–45.

[33] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2009, pp. 737–742.

[34] X. Guo, E. Ipek, and T. Soyata, "Resistive computation: Avoiding the power wall with low-leakage, STT-MRAM based computing," in *Proceedings of the International Symposium on Computer Architecture*, 2010, pp. 371–382.

[35] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2011, pp. 79–84.

[36] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proceedings*

of the International Symposium on Low Power Electronics and Design, 2012, pp. 237–242.

[37] J. Li, C. J. Xue, and Y. Xu, "STT-RAM based energy-efficiency hybrid cache for CMPs," in *Proceedings of the International Conference on VLSI and System-on-Chip*, 2011, pp. 31–36.

[38] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Design exploration of hybrid caches with disparate memory technologies," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 3, pp. 15:1–15:34, Dec. 2010.

[39] J. Li, L. Shi, C. J. Xue, C. Yang, and Y. Xu, "Exploiting set-level write non-uniformity for energy-efficient NVM-based hybrid cache," in *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia*, 2011, pp. 19–28.

[40] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache," in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2012, pp. 109–118.

[41] Q. Li, J. Li, L. Shi, C. J. Xue, and Y. He, "MAC: Migration-aware compilation for STT-RAM based hybrid cache in embedded systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2012, pp. 351–356.

[42] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2012, pp. 191–196.

[43] C. W. Smullen, IV, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011, pp. 50–61.

[44] Z. Sun, X. Bi, H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu, "Multi retention level STT-RAM cache designs with a dynamic refresh scheme," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 329–338.

[45] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile STT-RAM caches for enhanced performance

in CMPs," in *Proceedings of the Design Automation Conference*, 2012, pp. 243–252.

[46] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STTRAM scaling and retention failure," *Intel Technology Journal*, vol. 17, no. 1, pp. 54–75, May 2013.

[47] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 2–13.

[48] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2010, pp. 136–141.

[49] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proceedings of the International Conference on Computer-Aided Design*, 2009, pp. 264–268.

[50] T. Zheng, J. Park, M. Orshansky, and M. Erez, "Variable-energy write STT-RAM architecture with bit-wise write-completion monitoring," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2013, pp. 229–234.

[51] C.-L. Yang and A. R. Lebeck, "Push vs. pull: Data movement for linked data structures," in *Proceedings of the International Conference on Supercomputing*, 2000, pp. 176–186.

[52] Y. Solihin, J. Lee, and J. Torellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings of the International Symposium on Computer Architecture*, 2002, pp. 171–182.

[53] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 4, pp. 448–463, Apr. 2005.

[54] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing-in-memory taxonomy and a case for studying fixed-function PIM," presented at the *Workshop on Near-Data Processing*, 2013.

[55] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proceedings of the International Symposium on Microarchitecture*, 2013, pp. 185–197.

[56] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *Proceedings of the International 3D Systems Integration Conference*, 2013, pp. 1–7.

[57] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *Proceedings of the High Performance Extreme Computing Conference*, 2013, pp. 1–6.

[58] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 2014, pp. 85–98.

[59] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," presented at the *Workshop on Near-Data Processing*, 2014.

[60] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015, pp. 283–295.

[61] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, Mar. 2015.

[62] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data access optimization in a

processing-in-memory system," in *Proceedings of the International Conference on Computing Frontiers*, 2015, pp. 6:1–6:8.

[63] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3D-stacked DRAM," in *Proceedings of the International Symposium on Computer Architecture*, 2015, pp. 131–143.

[64] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 117–128.

[65] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *Proceedings of the International Symposium on Computer Architecture*, 2008, pp. 453–464.

[66] P. Ranganathan, "From microprocessors to Nanostores: Rethinking data-centric systems," *IEEE Computer*, vol. 44, no. 1, pp. 39–48, Jan. 2011.

[67] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 485–498.

[68] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "AC-DIMM: Associative computing with STT-MRAM," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 189–200.

[69] Y. Wang, Y. Han, L. Zhang, H. Li, and X. Li, "ProPRAM: Exploiting the transparent logic resources in non-volatile memory for near data computing," in *Proceedings of Design Automation Conference*, 2015, pp. 47:1–47:6.

[70] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proceedings of the International Symposium on Microarchitecture*, 2009, pp. 347–357.

[71] J. Ahn and K. Choi, "LASIC: Loop-aware sleepy instruction caches based on STT-RAM technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 1197–1201, May 2014.

[72] Y. Li, D. Parikh, Y. Zhang, K. Sankaranarayanan, M. Stan, and K. Skadron, "State-preserving vs. non-state-preserving leakage control in caches," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2004, pp. 22–27.

[73] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999, pp. 267–269.

[74] H. Sato and T. Sato, "A static and dynamic energy reduction technique for I-cache and BTB in embedded processors," in *Proceedings of the Asia and South Pacific Design Autmoation Conference*, 2004, pp. 831–834.

[75] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Exploiting fixed programs in embedded systems: A loop cache example," *IEEE Computer Architecture Letters*, vol. 1, no. 1, p. 2, Jan.–Dec. 2002.

[76] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[77] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000.

[78] A. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research," *IEEE Computer Architecture Letters*, vol. 1, no. 1, p. 7, Jan.–Dec. 2002.

[79] H. Sun, C. Liu, W. Xu, J. Zhao, N. Zheng, and T. Zhang, "Using magnetic RAM to build low-power and soft error-resilient L1 cache," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 19–28, Jan. 2012.

[80] H. Homayoun, A. Sasan, A. V. Veidenbaum, H.-C. Yao, S. Golshan, and P. Heydari, "MZZ-HVS: Multiple sleep modes zig-zag horizontal and vertical sleep transistor sharing to reduce leakage power in on-chip SRAM peripheral circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 12, pp. 2303–2316, Dec. 2011.

[81] J. Ahn and K. Choi, "Lower-bits cache for low power STT-RAM caches," in *Proceedings of the International Symposium on Circuits and Systems*, 2012, pp. 480–483.

[82] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Laboratories, Tech. Rep. HPL-2008-20, 2008.

[83] J. Ahn, S. Yoo, and K. Choi, "Write intensity prediction for energy-efficient non-volatile caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2013, pp. 223–228.

[84] ——, "Prediction hybrid cache: An energy-efficient STT-RAM cache architecture," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 940–951, Mar. 2016.

[85] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy, "Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture," in *Proceedings of the Design Automation Conference*, 2012, pp. 492–497.

[86] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L^3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2013, pp. 143–154.

[87] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, Apr. 2008.

[88] S. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proceedings of the International Symposium on Microarchitecture*, 2010, pp. 175–186.

[89] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 430–441.

[90] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *Proceedings of the International Symposium on Microarchitecture*, 2012, pp. 235–246.

[91] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proceedings of the International Symposium on Computer Architecture*, 2006, pp. 167–178.

[92] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the International Symposium on Microarchitecture*, 2006, pp. 423–432.

[93] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal bypass monitor for high performance last-level caches," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 315–324.

[94] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, "An energy efficient cache design using spin torque transfer (STT) RAM," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2010, pp. 389–394.

[95] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[96] *2Gb: x4, x8, x16 DDR3 SDRAM*, Micron Technology, 2006.

[97] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the International Symposium on Computer Architecture*, 2000, pp. 128–138.

[98] D. S. Gracia, G. Dimitrakopoulos, T. M. Arnal, M. G. H. Katevenis, and V. V. Yúfera, "LP-NUCA: Networks-in-cache for high-performance low-power embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 8, pp. 1510–1523, Aug. 2012.

[99] Y. Chen, X. Wang, H. Li, H. Xi, Y. Yan, and W. Zhu, "Design margin exploration of spin-transfer torque RAM (STT-RAM) in scaled technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 12, pp. 1724–1734, Dec. 2010.

[100] *Calculating Memory System Power for DDR3*, TN-41-01, Micron Technology, 2007.

[101] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[102] A. Phansalkar, A. Joshi, and L. K. John, "Subsetting the SPEC CPU2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 69–76, Mar. 2007.

[103] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation," in *Proceedings of the International Symposium on Microarchitecture*, 2004, pp. 81–92.

[104] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[105] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 234–244.

[106] J. Ahn, S. Yoo, and K. Choi, "DASCA: Dead write prediction assisted STT-RAM cache architecture," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014, pp. 25–36.

[107] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings of the International Symposium on Computer Architecture*, 2001, pp. 144–154.

[108] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture*, 2010, pp. 60–71.

[109] Z. Wang, S. M. Khan, and D. A. Jiménez, "Improving writeback efficiency with decoupled last-write prediction," in *Proceedings of the International Symposium on Computer Architecture*, 2012, pp. 309–320.

[110] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 14–23.

[111] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the International Symposium on Computer Architecture*, 1990, pp. 364–373.

[112] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.

[113] J. Wang, X. Dong, and Y. Xie, "OAP: An obstruction-aware cache management policy for STT-RAM last-level caches," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2013, pp. 847–852.

[114] J. Ahn, S. Yoo, and K. Choi, "Dynamic power management of off-chip links for hybrid memory cubes," in *Proceedings of the Design Automation Conference*, 2014, pp. 139:1–139:6.

[115] ——, "Low-power hybrid memory cubes with link power management and two-level prefetching," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 453–464, Feb. 2016.

[116] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 145–156.

[117] M. J. Khurshid and M. Lipasti, "Data compression for thermal mitigation in the hybrid memory cube," in *Proceedings of the International Conference on Computer Design*, 2013, pp. 185–192.

[118] A. Athavale and C. Christensen, *High-Speed Serial I/O Made Simple*. San Jose, CA: Xilinx Connectivity Solutions, 2005.

[119] V. Soteriou and L.-S. Peh, "Dynamic power management for power optimization of interconnection networks using on/off links," in *Proceedings of the Symposium on High Performance Interconnects*, 2003, pp. 15–20.

[120] ——, "Design-space exploration of power-aware on/off interconnection networks," in *Proceedings of the International Conference on Computer Design*, 2004, pp. 510–517.

[121] M. Alonso, J.-M. Martinez, V. Santonja, and P. Lopez, "Power saving in regular interconnection networks built with high-degree switches," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2005, p. 5b.

[122] E. J. Kim, G. M. Link, K. H. Yum, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and C. R. Das, "A holistic approach to designing energy-efficient cluster interconnects," *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 660–671, Jun. 2005.

[123] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the International Symposium on Microarchitecture*, 2009, pp. 316–326.

[124] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proceedings of the International Symposium on Microarchitecture*, 2006, pp. 397–408.

[125] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting midway: Improving CMP performance with memory-side prefetching," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 289–298.

[126] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2012, pp. 33–38.

[127] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the International Symposium on Microarchitecture*, 2009, pp. 469–480.

[128] J. Poulton, R. Palmer, A. M. Fuller, T. Greer, J. Eyles, W. J. Dally, and M. Horowitz, "A 14-mW 6.25-Gb/s transceiver in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 12, pp. 2745–2757, Dec. 2007.

[129] K. Fukuda, H. Yamashita, G. Ono, R. Nemoto, E. Suzuki, T. Takemoto, F. Yuki, and T. Saito, "A 12.3mW 12.5Gb/s complete transceiver in 65nm CMOS," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2010, pp. 368–369.

[130] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk memory access prediction and streaming," in *Proceedings of the International Symposium on Microarchitecture*, 2014, pp. 545–557.

[131] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 37–48.

[132] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 261–272.

[133] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the International Symposium on Microarchitecture*, 2013, pp. 468–479.

[134] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 249–260.

[135] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for memcached," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 36–47.

[136] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 24–35.

[137] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 349–362.

[138] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the Symposium on Operating System Design and Implementation*, 2012, pp. 17–30.

[139] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2013, pp. 22:1–22:12.

[140] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[141] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "KLA: A new algorithmic paradigm for parallel graph computations," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2014, pp. 27–38.

[142] ARM Cortex-A5 Processor. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a5.php

[143] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device," in *Proceedings of the International Symposium on Microarchitecture*, 2013, pp. 198–209.

[144] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 237–248.

[145] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[146] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 253–264.

[147] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.

[148] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995.

[149] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying scalable graph processing with a domain-specific language," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2014, pp. 208–218.

[150] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the International Symposium on Computer Architecture*, 1981, pp. 81–87.

[151] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," in *Proceedings of the International Conference on World Wide Web*, 1998, pp. 107–117.

[152] Laboratory for Web Algorithmics. [Online]. Available: http://law.di.unimi.it/datasets.php

[153] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the Conference on Internet Measurement*, 2007, pp. 29–42.

[154] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, Aug. 1998.

[155] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2015, pp. 336–348.

[156] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2000, pp. 161–171.

[157] T. L. Sterling and H. P. Zima, "Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing," in *Proceedings of the Conference on Supercomputing*, 2002, p. 48.

[158] S. Thoziyoor, J. Brockman, and D. Rinzler, "PIM lite: A multithreaded processor-in-memory prototype," in *Proceedings of the Great Lakes Symposium on VLSI*, 2005, pp. 64–69.

[159] Y. Solihin, J. Lee, and J. Torrellas, "Automatic code mapping on an intelligent memory architecture," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1248–1266, Nov. 2001.

[160] D. Keen, M. Oskin, J. Hensley, and F. T. Chong, "Cache coherence in intelligent memory systems," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 960–966, Jul. 2003.

[161] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, Mar. 2007.

[162] J. Leskovec and A. Krevl. (2014, Jun.) SNAP Datasets: Stanford large network dataset collection. [Online]. Available: http://snap.stanford.edu/data

[163] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel® AVX: New frontiers in performance improvements and energy efficiency," Intel Corporation, Tech. Rep., May 2008.

[164] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker, "Active memory operations," in *Proceedings of the International Conference on Supercomputing*, 2007, pp. 232–241.

[165] Z. Fang, L. Zhang, J. B. Carter, S. A. McKee, A. Ibrahim, M. A. Parker, and X. Jiang, "Active memory controller," *Journal of Supercomputing*, vol. 62, no. 1, pp. 510–549, Oct. 2012.

[166] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.

[167] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proceedings of the International Conference on Data Mining*, 2009, pp. 229–238.

[168] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *Proceedings of the International Conference on Data Engineering*, 2013, pp. 362–373.

[169] R. Narayanan, B. Özışıkyılmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A benchmark suite for data mining workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2006, pp. 182–188.

[170] E. Cooper-Balis, P. Rosenfeld, and B. Jacob, "Buffer-on-board memory systems," in *Proceedings of the International Symposium on Computer Architecture*, 2012, pp. 392–403.

[171] "Intel® C102/C104 scalable memory buffer datasheet," Intel, Feb. 2014.

[172] J. Friedrich, H. Le, W. Starke, J. Stuechli, B. Sinharoy, E. J. Fluhr, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, D. Hogenmiller, F. Malgioglio, R. Nett, R. Puri, P. Restle, D. Shan, Z. T. Deniz, D. Wendel, M. Ziegler, and D. Victor, "The POWER8™ processor: Designed for big data, analytics, and cloud environments," in *Proceedings of the International Conference on IC Design & Technology*, 2014, pp. 1–4.

[173] Synopsys DesignWare Library – Datapath and Building Block IP. [Online]. Available: http://www.synopsys.com/dw/buildingblock.php

[174] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon, "Demystifying 3D ICs: The pros and cons of going vertical," *IEEE Design & Test of Computers*, vol. 22, no. 6, pp. 498–510, Nov./Dec. 2005.

[175] J. Ahn, S. Yoo, and K. Choi, "AIM: Energy-efficient aggregation inside the memory hierarchy," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 34:1–34:24, Oct. 2016.

[176] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, Jul.-Dec. 2015.

[177] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[178] M. J. Garzarán, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas, "Architectural support for parallel reductions in scalable shared-memory multiprocessors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp. 243–254.

[179] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *Proceedings of the International Symposium on Microarchitecture*, 2015, pp. 13–25.

[180] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," The University of Texas at Austin, Tech. Rep. TR-HPS-2010-002, Apr. 2010.

[181] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *Proceedings of the International Symposium on Computer Architecture*, 2010, pp. 72–82.

[182] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The dirty-block index," in *Proceeding of the International Symposium on Computer Architecture*, 2014, pp. 157–168.

[183] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[184] J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-add in data parallel architectures," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005, pp. 132–142.

[185] C. Rohkohl, B. Keck, H. Hofmann, and J. Hornegger, "RabbitCT - an open platform for benchmarking 3D cone-beam reconstruction algorithms," *Medical Physics*, vol. 36, no. 9, pp. 3940–3944, Sep. 2009.

[186] L. A. Feldkamp, L. C. Davis, and J. W. Kress, "Practical cone-beam algorithm," *Journal of the Optical Society of America A*, vol. 1, no. 6, pp. 612–619, Jun. 1984.

[187] Y. Sadd, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[188] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation," in *Proceedings of the International Symposium on Computer Architecture*, 2014, pp. 349–360.

[189] K.-N. Lim, W.-J. Jang, H.-S. Won, K.-Y. Lee, H. Kim, D.-W. Kim, M.-H. Cho, S.-L. Kim, J.-H. Kang, K.-W. Park, and B.-T. Jeong, "A 1.2V 23nm 6F^2 4Gb

DDR3 SDRAM with local-bitline sense amplifier, hybrid LIO sense amplifier and dummy-less array architecture," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2012, pp. 42–44.

[190] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the Neural Information Processing Systems*, 2012, pp. 1097–1105.

[191] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

요약

메모리 시스템은 현대 컴퓨터 시스템의 성능 및 에너지 효율을 결정하는 중요한 부분 중 하나이다. 특히 최근 몇 년 간 멀티프로세서와 같이 계산 효율을 개선할 수 있는 새로운 컴퓨터 구조가 제안되고 대용량의 메모리에서 높은 대역폭으로 데이터를 읽고 처리하는 데이터 집약적인 어플리케이션의 중요성이 대두되면서 메모리가 병목이 되는 경향은 점점 더 심화되고 있다. 이러한 '메모리 벽' 문제를 해결하기 위해 본 논문은 최근 개발된 새로운 메모리 기술의 장단점을 분석하고 이를 잘 활용할 수 있는 고성능 저전력 메모리 계층을 설계하는 것을 목표로 한다.

본 논문의 전반부는 새로운 비휘발성 메모리의 일종인 STT-RAM을 이용해 저전력 온칩 캐시를 설계하는 방법에 초점을 맞춘다. STT-RAM은 SRAM이나 eDRAM과 같은 전하 기반 메모리와 비교했을 때 비휘발성, 낮은 대기 전력 소모 및 높은 집적도를 제공한다는 장점이 있다. 반면 STT-RAM은 SRAM보다 데이터를 쓰는 데에 시간 및 에너지를 더 소모한다는 단점이 있는데, 이로 인해 기존의 SRAM 기반 캐시를 단순히 STT-RAM으로 대체하기만 하면 오히려 캐시의 에너지 소모가 증가하는 것으로 나타난다.

이러한 문제를 해결하기 위해 본 논문에서는 STT-RAM의 쓰기 동작이 시스템의 성능 및 에너지 소모에 미치는 영향을 최소화하는 네 가지 새로운 컴퓨터 구조 기술을 제안한다. 첫째, 상대적으로 쓰기 동작이 드문 명령어 캐시에 STT-RAM을 적용하고 STT-RAM의 비휘발성을 활용해 짧은 반복문을 실행하는 중에 STT-RAM 명령어 캐시를 꺼서 에너지 소모를 줄이는 LASIC이라는 방법을 제안한다. 둘째, 많은 어플리케이션 데이터가 상위 비트보다 하위 비트의 값이 자주 변경된다는 점을 활용해서 데이터의 하위 비트만 작은 SRAM 캐시에 저장하는 하위 비트 캐시(Lower-Bits Cache)라는 방법을 제안한다. 셋째, SRAM과 STT-RAM을 모두 사용하는 혼성 캐시에서 캐시 블록을 할당할 때 쓰기 동작이 많을 것으로 예측되는 캐시

블록을 SRAM에 할당하는 예측 기반 혼성 캐시(Prediction Hybrid Cache)를 제안한다. 넷째, 캐시를 우회하더라도 캐시 미스를 증가시키지 않는 쓰기 동작을 정의 및 예측하고 이러한 쓰기 동작이 캐시를 우회하도록 하는 새로운 캐시 구조인 DASCA를 제안한다.

본 논문의 후반부는 논리 회로가 통합된 DRAM을 기반으로 지능적인 메인 메모리 및 이를 지원하는 호스트 구조를 설계하는 방법을 다룬다. 오늘날의 컴퓨터 구조에서 메인 메모리는 데이터를 저장하는 단순한 용도로만 사용되는데, 그 이유는 DRAM으로 구현된 메인 메모리에 데이터를 처리하는 기능을 직접 구현하는 것이 제조 비용을 크게 증가시키기 때문이었다. 그러나 최근 3D 적층 기술이 발전하면서 복잡한 논리 회로를 메인 메모리에 저비용으로 구현하는 것이 가능해졌으며, 이를 통해 메인 메모리와 CPU 사이의 대역폭 병목에 제한받지 않고 메모리 안에서 계산을 수행하는 프로세싱 인 메모리(PIM) 등과 같은 지능적인 메인 메모리를 현실적으로 구현할 수 있는 가능성이 열리게 되었다. 이러한 새로운 기술을 잘 활용하기 위해서는 어떤 기능을 메인 메모리에 구현하는 것이 최선인지, 그리고 이러한 새로운 기능을 기존 시스템에 어떻게 노출시킬 것인지 정하는 것이 중요하다.

이를 위해 본 논문에서는 논리 회로가 통합된 DRAM을 활용해 시스템의 성능 및 에너지 효율을 개선하는 네 가지 새로운 컴퓨터 구조를 제안한다. 첫째, HMC(Hybrid Memory Cube)라는 상용화된 논리 회로 통합 DRAM의 논리 회로 층을 활용해 실제 대역폭 소모량에 맞추어 HMC 링크의 일부를 동적으로 끄는 방법 및 메모리 안에 프리페치 버퍼를 추가해 링크 대역폭을 소모하지 않고도 공격적으로 프리페치하는 방법을 제안한다. 둘째, 대용량 그래프 처리라는 중요한 어플리케이션을 메모리 안에서 수행해서 스케일러블한 성능을 제공하는 Tesseract라는 가속기 구조를 제안한다. 셋째, PIM 연산을 캐시 일관성 및 가상 메모리를 지원하는 CPU 명령어의 형태로 기존 시스템에 제공하는 방식을 통해 PIM을 기존 시스템에 저비용으로 구현하는 PEI(PIM-Enabled Instruction)를 제안한다. 넷째, PIM 기술로 시스템의 에너지 효율을 높이기 위해 PIM 연산을 메모리 계층의 어디에서나 수행할 수 있도록 하는 하드웨

어와 기존 어플리케이션을 자동으로 PIM을 사용할 수 있도록 변환하는 컴파일러를 제공하는 AIM(Aggregation-in-Memory)을 제안한다.

주요어: 컴퓨터 구조, 메모리 계층, 비휘발성 메모리, STT-RAM, 논리 회로가 통합된 DRAM, 프로세싱 인 메모리
학번: 2011-20873